Deep reinforcement learning for the vehicle routing problem

Ali Yaddaden

THÈSE POUR OBTENIR LE GRADE DE DOCTEUR
DE L'INSTITUT MINES-TÉLÉCOM (IMT) -
ÉCOLE NATIONALE SUPÉRIEURE DES MINES D'ALÈS (IMT MINES ALÈS)

En Informatique

École doctorale : I2S
Portée par l'Université de Montpellier

Unité de recherche : EuroMov DHM

Deep reinforcement learning for the Vehicle Routing Problem

Présentée par Ali Yaddaden

Le 06 novembre 2023

Sous la direction de Michel Vasquez
et l'encadrement de Sébastien Harispe

Devant le jury composé de :

Christine SOLNON, Professeure, INSA Lyon, CITI Lab	Présidente
Jin-Kao HAO, Professeur, Université d'Angers, LERIA	Rapporteur
El-Ghazali TALBI, Professeur, Université de Lille, CRISTAL	Rapporteur
Rodolphe GIROUDEAU, Maître de Conférences HDR, Université de Montpellier, LIRMM	Examinateur
Audrey DUPONT, Senior R&D Software Engineer, PROS	Invitée
Michel VASQUEZ, Enseignant-Chercheur HDR, IMT Mines Alès, Euromov DHM	Directeur
Sébastien HARISPE, Maître-assistant, IMT Mines Alès, Euromov DHM	Encadrant

COLLÈGE DOCTORAL
UNIVERSITÉ DE MONTPELLIER

IMT Mines Alès
École Mines-Télécom

Abstract

Recent advances in deep and reinforcement learning have led to breakthroughs in several fields (e.g. game of Go, protein structure prediction, ChatGPT), in some contexts achieving superhuman performance. The progress made in these fields of Artificial Intelligence naturally leads us to study approaches based on neural networks trained by reinforcement learning for tackling complex combinatorial problems. Today, many of these problems are, more or less effectively and satisfactorily, dealt with by solving approaches relying on time-consuming definition proposed by experts in the field of Operations Research.

In this context, this thesis focuses on the study of deep and reinforcement learning for the solution of two combinatorial optimization problems known as vehicle routing problems (VRP): the Capacitated Vehicle Routing Problem (CVRP), and the Ride-Hailing Problem (RHP). The former is a well-known classical optimization problem, while the latter is a more recent variant involving uncertainty, e.g. trip duration. Our main line of research focuses on the study of solvers based on deep neural networks trained using reinforcement learning (policy gradient and Deep Q-learning algorithms) on large datasets of unsolved instances. Among other things, these solvers make it possible to overcome the need for manual definition of solving methods and delegate this task to a deep neural network. For example, the network will estimate a conditional probability that is useful for iteratively constructing a candidate solution, e.g. the probability that visiting a specific client given a list of previously visited customers and the problem configuration, will bring us closer to the optimal solution. More specifically, we are studying neural

network architectures based on the attention mechanism. The latter makes our solvers agnostic to instance size, enabling us to empirically study their generalization capacity, particularly in their reuse on VRP instances of a different nature or size from those considered during the training phases.

This manuscript is structured around three contributions. The first one aims at studying the contribution of transfer learning to the resolution of combinatorial optimization problems using neural networks. Our study is based on the implicit transfer of knowledge from the traveling salesman problem (TSP) to the CVRP. The aim is to study whether a model trained to solve a given VRP problem can be used to solve another similar problem following the application of a few additional training steps. In the second contribution, we propose a new two-steps method involving deep neural networks and a shortest path algorithm to handle the capacity constraint. Through our various experiments, we demonstrate the competitiveness of this method with neural approaches in the literature, as well as with classical CVRP heuristics. In our final contribution, we study the contribution of deep neural network-based solution methods to a ride-hailing problem that includes a dimension of uncertainty (stochastic nature of request observation and travel time). We propose a neural approach based on reinforcement learning, capable of handling variable numbers of requests and vehicles. Our results demonstrate the effectiveness of such an approach in tackling this type of problem.

Keywords: Combinatorial optimization, Vehicle routing problem, Deep learning, Attention mechanism, Deep reinforcement learning, Policy gradient methods, Deep Q-learning.

Résumé

Les avancées récentes en apprentissage profond et par renforcement ont récemment conduit à des ruptures dans plusieurs domaines (e.g. jeu de Go, prédiction de structures protéiques, ChatGPT), atteignant dans certains contextes des performances suprahumaines. Les progrès amenés par ces domaines de l'Intelligence Artificielle invitent naturellement à étudier les approches à base de réseaux de neurones entraînés par renforcement pour aborder des problèmes complexes à forte combinatoire. Aujourd'hui, nombre de ces problèmes sont traités, de manière plus ou moins efficace et satisfaisante, par des approches de résolution reposant sur une définition chronophage proposée par des experts du domaine de la recherche opérationnelle.

Dans ce contexte, cette thèse se concentre sur l'étude de l'apprentissage profond et par renforcement pour la résolution de deux problèmes d'optimisation combinatoire dits de tournées de véhicules (VRP) : le problème de tournées de véhicules avec contraintes de capacités (CVRP), et le problème de covoiturage (RHP). Le premier est un problème d'optimisation classique bien connu, tandis que le second est une variante plus récente impliquant de l'incertitude, e.g. durée des trajets. Notre ligne principale de recherche se concentre sur l'étude de solveurs reposant sur des réseaux de neurones profonds entraînés à l'aide de l'apprentissage par renforcement (algorithmes de type *policy gradient* et *Deep Q-learning*) sur de vastes jeux de données d'instances non résolues. Ces solveurs permettent entre autres de s'affranchir de la définition manuelle de méthodes de résolution et de déléguer cette tâche à un réseau de neurones profond. Le réseau estimera, par exemple,

une probabilité conditionnelle utile pour la construction itérative d'une solution candidate, e.g. la probabilité que la visite d'un client spécifique, sachant une liste de clients déjà visités et la configuration du problème, nous approche de la tournée optimale. Nous étudions plus précisément dans nos travaux des architectures de réseaux de neurones basées sur le mécanisme d'attention. Ce dernier rend nos solveurs agnostiques à la taille des instances, ce qui nous permet d'étudier empiriquement leur capacité de généralisation, en particulier dans leur réutilisation sur des instances de VRP de nature ou de taille différentes de celles considérées lors des phases d'entraînement.

Ce manuscrit est structuré autour de trois contributions. La première vise à étudier l'apport de l'apprentissage par transfert dans le cadre de la résolution de problèmes d'optimisation combinatoire par réseaux de neurones. Nous nous basons dans notre étude sur le transfert implicite de connaissances du problème de voyageur de commerce (TSP) vers le CVRP. L'objectif est d'étudier si un modèle entraîné pour résoudre un problème de VRP donné, peut être utilisé pour résoudre un autre problème similaire suite à l'application de quelques étapes d'entraînement supplémentaires. Dans la deuxième contribution, nous proposons une nouvelle méthode à deux phases impliquant des réseaux de neurones profonds et un algorithme de plus court chemin pour gérer la contrainte de capacité. Nous montrons à travers nos différentes expérimentations la compétitivité de cette méthode avec les approches neuronales de la littérature ainsi que les heuristiques classiques du CVRP. Pour notre dernière contribution, nous étudions l'apport des méthodes de résolution à base de réseaux de neurones profonds pour un problème de covoiturage incluant une dimension d'incertitude (caractère stochastique de l'observation des requêtes et de la durée de trajet). Nous proposons pour cela une approche neuronale à base d'apprentissage par renforcement, capable de traiter des nombres variables de requêtes et de véhicules. Nos résultats montrent l'efficacité d'une telle approche pour aborder ce type de problèmes.

Mots clés : Optimisation combinatoire, Problème de tournées de véhicules, Apprentissage profond, Mécanisme d'attention, Apprentissage par renforcement profond, *Policy-gradient*, *Deep Q-learning*.

Résumé étendu

Introduction

La Recherche Opérationnelle et l'Optimisation Combinatoire jouent un rôle essentiel dans les industries modernes. En effet, ces deux disciplines permettent de résoudre des problèmes dans plusieurs secteurs tels que la logistique, la planification, l'énergie, la finance, les télécommunications, etc. À travers des outils mathématiques et algorithmiques, ces deux domaines visent à trouver les solutions parmi un ensemble fini, souvent vaste, de configurations possibles, tout en tenant compte des contraintes et des objectifs.

Le problème de tournées de véhicules est l'une des classes de problèmes de la Recherche Opérationnelle et de l'Optimisation Combinatoire les plus importantes et les plus répandues [184]. Ce problème vise à optimiser les itinéraires pour un ensemble de véhicules qui doivent desservir un ensemble de clients, tout en respectant des contraintes telles que la capacité du véhicule, des horaires de livraison, etc. Ce problème est complexe à résoudre, car il implique une combinaison exponentielle de configurations possibles, ce qui rend impossible une énumération exhaustive pour des instances de taille supérieure à quelques dizaines de clients. Plus précisément, ce problème est $\mathcal{NP}$-difficile [112], ce qui implique qu'il n'existe pas à ce jour un algorithme efficace capable de résoudre en temps polynomial toutes les instances du problème. De plus, certaines variantes du problème impliquent de l'incertitude (e.g. sur les temps de trajets), ce qui rajoute de la difficulté lors de la résolution [146].

Les méthodes classiques de résolution du problème de tournées de véhicules incluent les méthodes exactes, les heuristiques et les métaheuristiques. Parmi les méthodes exactes, nous pouvons citer la programmation dynamique, la méthode de séparation et évaluation (*branch and bound*) et la programmation linéaire en nombres entiers, notamment les méthodes de type *branch-and-cut-and-price* réputées performantes sur ce problème [35, 56, 7, 2, 145, 74]. Cependant, celles-ci présentent des temps d'exécution importants qui font qu'elles ne peuvent pas être utilisées pour résoudre des instances de très grande taille. Les méthodes heuristiques, quant à elles, trouvent rapidement des solutions candidates, sans garantie d'optimalité. Nous pouvons distinguer deux catégories : les heuristiques constructives et les heuristiques d'amélioration. Dans la première catégorie, nous retrouvons l'heuristique de *Clarke & Wright* [36], l'heuristique du plus proche voisin [174], et les méthodes à deux phases *Cluster-first Route-second* [64] et *Route-first Cluster-second* [17]. La deuxième catégorie comprend des algorithmes tels que λ-OPT, SWAP, RELOCATE, MOVE. Les métaheuristiques sont des méthodes plus génériques, capables de traiter différents problèmes d'optimisation [173]. Ces méthodes alternent entre des phases d'intensification de la recherche autour d'un voisinage, et de diversification pour échapper aux optima locaux. Plusieurs métaheuristiques ont été utilisées pour résoudre les problèmes de tournées telles que le recuit simulé [136], la recherche taboue [62], les algorithmes génétiques [149], etc.

Récemment, des avancées importantes ont été effectuées en apprentissage automatique, notamment en apprentissage profond et par renforcement. L'apprentissage profond repose sur l'utilisation de réseaux de neurones profonds comportant plusieurs couches de neurones interconnectés [65]. Contrairement aux techniques classiques d'apprentissage automatique, l'apprentissage profond vise à apprendre à extraire automatiquement les caractéristiques d'intérêt pour la résolution de la tâche d'apprentissage, à partir des données d'entraînement [109]. Cette dernière décennie a vu l'émergence de nouvelles architectures de réseaux de neurones profonds tels que les réseaux de neurones récurrents GRU (*Gated Recurrent Units*) [31], les *Transformers* [178], les réseaux de neurones sur graphe [194]. De plus, des techniques de régularisation, telle que la normalisation des couches (*layer normalization*) [10] et la connexion de saut (*skip connection*) [72], ont contribué

grandement à stabiliser l'entraînement des réseaux de neurones, et ainsi permettre d'entraîner des modèles plus profonds. En outre, de nouveaux algorithmes basés sur l'algorithme de descente de gradient ont été proposés (e.g. Adam [96]) et améliorent la convergence des réseaux de neurones lors de l'entraînement.

L'apprentissage par renforcement est une branche de l'apprentissage automatique qui vise à permettre à un agent d'apprendre une stratégie (politique) de prise des décisions (actions) à travers l'observation de l'état dans lequel il se trouve dans un environnement afin d'optimiser une fonction objectif (généralement, maximiser une fonction de récompense) [171]. Cette interaction entre l'agent et l'environnement est modélisée via un processus de décision markovien. En apprentissage par renforcement, de nouveaux algorithmes ont aussi été proposés. Ceux-ci se basent sur des réseaux de neurones profonds pour approximer les fonctions d'évaluation d'état-action ou les politiques. Nous parlons alors d'apprentissage par renforcement profond [71]. Nous pouvons citer, par exemple, l'algorithme de *Deep Q-Learning* [130] qui utilise un réseau de neurones profond pour estimer la fonction d'état-action, et ainsi évaluer dans un état donné la valeur de la récompense espérée à long terme pour chaque action. Par ailleurs, des réseaux de neurones profonds sont aussi utilisés dans des algorithmes de type *policy gradient* (e.g. REINFORCE [192]) pour estimer directement la politique de l'agent.

Grâce à ces avancées, des ruptures dans des domaines réputés difficiles et complexes à traiter par ordinateur ont été observées. L'apprentissage profond et par renforcement ont permis d'atteindre des performances inégalées en traitement d'image [72], traitement du langage naturel [119], résolution de jeux combinatoires tels que les échecs ou le Go [166]. L'un des progrès les plus remarquables est l'introduction d'agents conversationnels (e.g. ChatGPT) capables de dialoguer et de se souvenir des messages d'une conversation [137]. En outre, elles ont ouvert de nouvelles opportunités et ont notamment ravivé l'intérêt de l'utilisation des réseaux de neurones pour la résolution de problèmes d'optimisation combinatoire [185]. Bien que nous fassions mention de diverses approches de résolution pour ces problèmes, celles-ci reposent principalement sur une définition manuelle de stratégies de résolution. En plus d'être chronophage, cette définition manuelle requiert souvent une expertise dans le domaine afin d'aboutir à des stratégies intéressantes en termes de performance et de temps d'exécution.

Dans cette thèse, nous proposons l'étude de solveurs à base d'apprentissage profond et par renforcement afin de s'affranchir de la définition manuelle de stratégies de résolution pour des problèmes de tournées de véhicules. Pour cela, nous structurons ce manuscrit autour de trois contributions que nous détaillons dans la section suivante.

Contributions

Notre ligne principale de recherche se concentre sur l'étude de solveurs reposants sur des réseaux de neurones profonds entraînés à l'aide de l'apprentissage par renforcement sur de vastes jeux de données d'instances non résolues [20]. Ainsi, selon la classification de Bengio et al. [21], nos travaux se situent dans la catégorie des méthodes d'apprentissage de bout-en-bout pour la résolution de problèmes d'optimisation combinatoire. En conséquence, nous n'abordons pas, par exemple, l'utilisation des méthodes d'apprentissage pour la définition des paramètres des algorithmes de résolution.

État de l'art et évaluations empiriques

Nous avons consacré une partie du travail de cette thèse à faire une synthèse de l'état des connaissances dans le domaine de l'apprentissage profond et par renforcement pour les problèmes de tournées. Nous avons examiné 35 contributions parues de 2015 à 2022 dans des conférences et des revues majeures dans les domaines de l'apprentissage automatique et de l'optimisation combinatoire. Le cadre de résolution introduit est connu sous le nom *cadre d'optimisation combinatoire neuronale* (*Neural Combinatorial Optimization framework (NCO)*) [20]. De plus, nous avons concentré notre revue sur le problème du voyageur de commerce (TSP) [6] étant donné que c'est le premier problème traité dans ce cadre de résolution, et sur le problème de tournées de véhicules avec contraintes de capacité (CVRP) [175]. Ainsi, nous avons constaté que la majorité des approches plébiscitent l'entraînement des modèles neuronaux via l'apprentissage par renforcement. Ce type d'apprentissage présente l'avantage d'entraîner via une succession d'essais et erreurs. L'autre paradigme d'entraînement utilisé est l'apprentissage supervisé,

qui nécessite de recourir à la démonstration de solutions optimales pour les instances utilisées durant l'entraînement. La collecte d'instances avec leurs solutions optimales peut être une tâche fastidieuse, voire impossible pour certaines variantes fortement contraintes du problème de tournée de véhicules. Ainsi, il devient avantageux d'utiliser l'apprentissage par renforcement.

En apprentissage supervisé, nous avons pu distinguer deux types de modèles : les modèles autoregressifs qui construisent la solution candidate de manière itérative [185], à l'image des heuristiques constructives, et les modèles non-autoregressifs conçus pour obtenir la solution en une seule fois, en produisant une carte de chaleur probabiliste sur la matrice d'adjacence [89]. En apprentissage par renforcement, nous distinguons les méthodes constructives qui partent d'une solution vide et y ajoute itérativement des nœuds jusqu'à ce qu'un critère d'arrêt soit vérifié (généralement, jusqu'à ce que tous les clients soient visités) [20, 100], des méthodes d'amélioration qui démarrent d'une solution initiale et cherchent à l'améliorer. Ces dernières intègrent les méthodes neuronales à l'intérieur d'une métaheuristique telle que la recherche à voisinage large, afin de lui déléguer la phase de construction de la solution candidate [80].

Nous constatons via notre état de l'art les différentes avancées en matière d'architecture de réseaux de neurones utilisées. Ainsi, les premiers modèles sont à base de réseaux de neurones récurrents, alors que les modèles récents et plus performants intègrent des *Transformers* et des réseaux de neurones sur graphe [100, 89]. Également, la majorité des modèles utilisent des architectures de type encodeur-décodeur [31]. Un premier réseau de neurones, appelé encodeur, est utilisé pour l'extraction automatique de caractéristiques d'intérêt pour la résolution. Son rôle est donc d'apprendre une représentation de l'instance qui facilitera sa résolution. Le second réseau de neurones, appelé décodeur, utilise la représentation fournie par l'encodeur afin de produire une solution candidate.

Une importante quantité de données est nécessaire pour entraîner les modèles. Des millions d'instances sont donc utilisées durant l'entraînement. Plus il y a d'instances dans le jeu de données d'entraînement, meilleure est la convergence des modèles. En outre, ces modèles sont entraînés sur des instances de tailles similaires, et généralement, des instances de petites tailles (20, 50 et 100 clients dans le cas du CVRP). Concernant les stratégies de recherche pour exploiter les mod-

èles, plusieurs ont été proposées, dont une recherche gloutonne, un échantillonnage stochastiques, une recherche par faisceau, une recherche arborescente Monte Carlo, etc.

Les protocoles d'évaluation des approches du cadre de l'optimisation combinatoire neuronale considèrent des instances générées aléatoirement par un échantillonnage de la loi uniforme sur le carré $[0, 1] \times [0, 1]$ [132]. Les résultats exposés dans la littérature montrent l'efficacité des modèles à base de réseaux de neurones profonds pour la résolution des problèmes de tournées de véhicules. Cependant, ces modèles tendent à être efficaces sur des instances de tailles semblables aux instances utilisées durant l'entraînement. Par exemple, un modèle entraîné sur des instances de 50 clients donnera de meilleurs résultats s'il est exploité sur des instances de 50 clients plutôt que s'il est exploité sur des instances de tailles inférieures ou supérieures. Ce protocole d'évaluation ne permet pas de saisir l'étendue des performances de ces modèles. En effet, pour être exhaustif, il faudrait considérer des instances de différentes tailles, ainsi que différentes répartitions des clients sur le plan, tel que le regroupement en clusters.

Dans l'optique d'avoir une meilleure idée des performances de ces modèles, la seconde partie de notre état de l'art est consacrée à l'évaluation empirique des performances des modèles d'optimisation combinatoire neuronale sur des instances hétérogènes. Pour cela, nous avons reproduit une architecture de l'état de l'art et nous l'avons évaluée sur des instances de CVRPLib [175]. Nos résultats montrent des performances du même ordre de grandeur que ceux rapportés dans la littérature sur des instances de petite taille, avec une répartition uniforme et en cluster des clients. Nous avons aussi constaté une baisse des performances pour des tailles d'instances plus grandes. Cependant, les performances de ces méthodes restent toujours inférieures à celles des métaheuristiques de l'état de l'art telle que HGS [182, 181].

Apprentissage par transfert pour les problèmes de tournées

Nous avons soulevé à travers l'état de l'art que l'entraînement des modèles nécessite une quantité très importante d'instances du problème considéré (de l'ordre du million). Cependant, il n'est pas aisé d'avoir accès à de telle quantité de données dans

des scénarios concrets. Nous proposons donc l'apprentissage par transfert dans le cadre de l'optimisation combinatoire neuronale. L'apprentissage par transfert est une approche d'apprentissage automatique où les connaissances acquises lors de la résolution d'une tâche peuvent être transférées et utilisées pour améliorer les performances dans une autre tâche apparentée [65]. Grâce aux connaissances préalables acquises, l'apprentissage par transfert peut accélérer le processus d'apprentissage pour la nouvelle tâche et améliorer les performances du modèle.

Notre cas d'étude porte sur le transfert implicite de connaissances d'un problème de tournées à un autre. Nous considérons dans ce cas le problème du voyageur de commerce (TSP) et le problème de tournées de véhicules avec contraintes de capacité (CVRP). Le CVRP étant une généralisation du TSP, ces deux problèmes constituent un cas d'étude idéal pour nos travaux. Nous considérons quelques hypothèses de départ : (1) la même architecture de réseau de neurones est utilisée pour les deux problèmes ; (2) tous les poids des neurones appris sur le TSP sont réutilisés sur le CVRP, il n'y a donc aucun poids réinitialisé aléatoirement ; (3) les mêmes caractéristiques sont utilisées pour encoder les deux problèmes ; (4) le modèle du CVRP est entraîné sur moins d'itérations. De plus, nous considérons quatre scénarios d'entraînement suivant la taille et la distribution des instances du TSP et du CVRP : (1) elles sont de la même taille et proviennent de la même distribution ; (2) elles sont de différentes tailles et proviennent de la même distribution ; (3) elles sont de la même taille et proviennent de différentes distributions ; (4) le TSP est utilisé comme tâche prétexte pour le CVRP. Les trois premiers scénarios correspondent à des cas d'apprentissage par transfert standard alors que le dernier scénario est un cas proche de l'apprentissage autosupervisé [122].

Nos différentes expériences indiquent que l'apprentissage par transfert peut être bénéfique dans les cas où relativement peu de données sont disponibles pour la tâche cible. Il améliore la performance asymptotique (finale) du modèle par rapport au modèle sans apprentissage de transfert tout en accélérant l'apprentissage sur quelques itérations. En outre, les modèles entraînés sur des instances de tailles similaires et provenant de la même distribution entre les tâches source (TSP) et cible (CVRP) donnent de meilleurs résultats. Enfin, dans les pires configurations, l'apprentissage par transfert ne semble pas nuire au processus d'apprentissage, étant donné les performances asymptotiques similaires observées.

Order-first Split-second neuronal pour le problème de tournées de véhicules avec contrainte de capacité

Cette contribution propose un nouvel algorithme à deux phases pour le problème de tournées de véhicules avec contrainte de capacité, suivant le même modèle que l'heuristique à deux phases *Route-first Cluster-second* [17]. Cette approche combine un réseau neuronal profond avec un algorithme de plus court chemin.

Le réseau de neurones profond est utilisé pour générer un ordre de visite des clients appelé tour géant. Ce dernier définit un graphe auxiliaire sur lequel un algorithme de plus court chemin est appliqué pour obtenir les tournées et leur coût associé (l'algorithme *Split*) [149, 150]. Cette approche combine les avantages de l'utilisation d'un réseau de neurones profond avec un algorithme efficace d'extraction de tournées. En effet, grâce au réseau de neurones, il n'est plus nécessaire de définir explicitement la méthode de construction du tour géant. De plus, la méthode bénéficie des avantages de *Split* qui extrait les meilleures tournées d'un tour géant. En outre, le coût des itinéraires renvoyés par *Split* nous permet d'entraîner le réseau neuronal à l'aide de l'apprentissage par renforcement.

Cette approche étant agnostique au modèle de réseau de neurones considéré, nous avons donc proposé et évalué trois modèles de réseaux neuronaux différents basés sur des réseaux de neurones sur graphe (GNN). En outre, nous avons proposé d'introduire l'angle polaire par rapport au dépôt en tant que nouvelle caractéristique brute pendant la phase d'encodage, ce qui s'est avéré particulièrement utile pour améliorer la convergence de notre modèle vers un meilleur optimum local. Nous proposons également, lors de la phase d'encodage, une définition du voisinage d'un nœud qui prend en compte la capacité du véhicule et les demandes des clients.

Lors de l'évaluation, nous avons constaté que le GNN basé sur le réseau de neurones *Transformer* donne les meilleurs résultats. De plus, nous constatons qu'une architecture peu profonde (GNN à trois couches) a une meilleure convergence qu'une architecture plus profonde (GNN à huit couches).

Nos résultats sont compétitifs par rapport à ceux des méthodes constructives basées sur des réseaux de neurones et des heuristiques. Notamment, notre modèle obtient de meilleurs résultats que *Route-first Cluster-second* qui est une heuristique

constructive de même nature que notre modèle. De plus, notre modèle est plus rapide pour l'apprentissage et l'inférence que les méthodes constructives neuronales. Nos évaluations sur CVRPLib confirment la performance de notre approche sur des instances de petite et moyenne tailles, indépendamment de la distribution des clients.

Apprentissage par renforcement profond pour le problème de covoiturage

Notre dernière contribution est consacrée à l'étude des approches d'optimisation combinatoire neuronales sur un problème stochastique et dynamique. Pour ce faire, nous avons choisi le problème du covoiturage (*Ride-hailing problem (RHP)*) [102], qui a gagné un intérêt significatif en raison des nombreuses applications proposant ce type de service. Il représente un excellent cas d'étude, car il comporte intrinsèquement des aspects dynamiques et stochastiques. En effet, il est impossible de connaître à l'avance la localisation de toutes les demandes des clients avant le début du processus de résolution du problème. De plus, il existe une incertitude concernant les temps de trajet des véhicules, qui dépendent des lieux de départ et d'arrivée ainsi que de l'heure de la journée. En outre, ce problème comporte deux aspects combinatoires : l'affectation des véhicules aux demandes et le repositionnement des véhicules pour anticiper les demandes futures et s'en rapprocher davantage pour assurer une meilleure qualité de service.

Pour résoudre ce problème, nous avons construit un réseau de neurones profond capable de traiter nombre fluctuant de demandes tout en manipulant une flotte variable de véhicules. En outre, nous avons conçu plusieurs stratégies de résolution. Dans la première stratégie, l'affectation des demandes aux véhicules, ainsi que le repositionnement des véhicules sont tous deux gérés par le réseau de neurones profond. Une deuxième stratégie consiste à gérer l'affectation des demandes aux véhicules à l'aide de réseaux de neurones profonds et à effectuer le repositionnement à l'aide d'un algorithme de plus proche voisin. Enfin, nous avons créé une stratégie qui se situe entre les deux premières. L'affectation des demandes aux véhicules est effectuée à l'aide de réseaux neuronaux profonds. Cependant, pour le repositionnement, le réseau neuronal profond prédit la zone à laquelle le véhicule

doit se rendre, et l'emplacement exact de repositionnement est sélectionné à l'aide d'un algorithme du plus proche voisin. Nous avons entraîné nos modèles à l'aide d'un algorithme de type *n-step dueling deep Q-learning* [189]. Pour les instances de petite taille, l'algorithme qui combine un réseau de neurones pour l'affectation des demandes et une approche de plus proche voisin pour le repositionnement est plus efficace. Pour les instances de plus grande taille, l'algorithme qui combine une approche de repositionnement par zone avec une approche de plus proche voisin est plus efficace. Toutefois, des améliorations sont encore possibles pour atteindre de meilleurs taux de satisfaction des demandes. Nos algorithmes satisfont en moyenne 40 % des demandes sur les petites instances et 68 % sur les grandes instances.

Notre approche présente l'avantage d'avoir de bonnes propriétés de généralisation. Ainsi, elle est capable de traiter à la fois des instances de la même taille que celles rencontrées lors de l'entraînement, mais également des instances de taille plus importante, sans qu'il soit nécessaire d'entraîner de nouveau le modèle sur la nouvelle taille de l'instance. En outre, nous avons observé de meilleurs résultats pour les politiques hybrides que pour les politiques basées sur l'apprentissage ou les politiques définies manuellement. Cela nous incite à étudier davantage l'hybridation entre les modèles basés sur l'apprentissage et les heuristiques définies manuellement.

Principaux apports

Nous pouvons résumer les principaux apports de cette thèse dans les points suivants :

- Un état de l'art approfondi de la littérature du cadre d'optimisation combinatoire neuronale, avec une classification des approches.

- Une évaluation empirique sur des benchmarks reconnus par la communauté des problèmes de tournées de véhicules. Celle-ci offre plus de détails sur les performances des modèles.

- La proposition de l'apprentissage par transfert pour le transfert de connaissances d'une politique de résolution du TSP vers une politique de résolution

du CVRP.

- Une nouvelle méthode à deux phases suivant le principe de *Route-first Split-second* pour la résolution du CVRP. Celle-ci se base sur la génération d'un ordre de visite des clients par réseau de neurones profond, et la délimitation des tournées via l'algorithme Split.
- La proposition de l'ajout de l'angle au dépôt comme nouvelle caractéristique pour l'encodage des instances du CVRP.
- La définition d'un réseau de neurones profond capable de traiter un nombre variable de clients avec un nombre variable de voitures pour le problème de covoiturage.
- La proposition d'algorithmes hybrides entre réseaux de neurones et algorithme du plus proche voisin pour le problème de covoiturage.

Conclusion et perspectives

Les travaux initiés dans cette thèse s'inscrivent dans les efforts de rapprochement entre la Recherche Opérationnelle et l'Intelligence Artificielle en étudiant l'application d'outils d'apprentissage automatique sur des problèmes d'optimisation combinatoire. Cette étude se fait à travers l'exploitation des récents développements en matière d'apprentissage profond et d'apprentissage par renforcement pour la résolution du problème de tournées de véhicules.

Cette thèse présente trois contributions principales dans le cadre de l'optimisation combinatoire neuronale. Un état de l'art a d'abord été réalisé afin de situer les approches de ce cadre par rapport à la littérature, que ce soit sur la méthodologie de résolution, ou sur les performances par rapport à des heuristiques et méta-heuristiques état de l'art pour le problème de tournées de véhicule. Ensuite, un premier travail sur l'apprentissage par transfert a été présenté afin de déterminer l'apport du transfert de connaissances d'un problème de tournées vers un autre. La troisième contribution présente un nouvel algorithme à deux phases permettant l'apprentissage d'une représentation implicite d'une solution d'un CVRP sous forme d'un tour géant. La dernière contribution présente un réseau de neurones

capable de traiter un nombre variable de requêtes avec un nombre variable de véhicules pour le problème de covoiturage.

À travers ces travaux, nous avons pu montrer que des méthodes à base de réseaux de neurones peuvent être efficaces pour la résolution de problèmes de tournées. Cependant, leurs performances restent encore en deçà de ce que proposent les métaheuristiques de l'état de l'art. Néanmoins, le cadre de l'optimisation combinatoire neuronale est encore récent. Il reste donc plusieurs pistes d'intérêt à explorer. Nous en proposons dans ce qui suit quelques-unes que nous avons identifiées, et nous les classons selon les critères suivants : la nature des instances, l'apprentissage des représentations, la nature algorithmique, le type de problème traité, et l'explicabilité.

Sur la nature des instances, nous avons observé que les modèles à base de réseaux de neurones sont efficaces sur des instances de petite à moyenne tailles. De plus, les méthodes hybrides sont capables de traiter de plus larges instances. Il y a donc un premier défi sur le traitement d'instances de plus grandes tailles. En outre, le nombre d'instances d'entraînement est conséquent. Il n'y a aucune garantie que de telles quantités de données soient disponibles pour des problèmes de tournées du monde réel, afin d'entraîner ces modèles. De même, l'aspect qualité des données n'a pas été considéré durant l'entraînement des modèles. Il n'est pas exclu qu'entraîner avec un jeu d'instances soigneusement sélectionnées donne de meilleurs modèles.

L'apprentissage de représentations a été abordé en proposant l'apprentissage par transfert, et en présentant différents encodeurs dans le cadre de l'apprentissage de représentations pour le tour géant. De futurs travaux peuvent considérer l'apprentissage autosupervisé [122], en s'inspirant des travaux en traitement d'image. Par exemple, en considérant l'apprentissage contrastif dans le but d'apprendre à différencier les instances similaires (e.g. à une rotation près) des instances différentes. De plus, il n'est pas exclu qu'il puisse exister d'autres architectures de réseaux de neurones plus adaptées aux problèmes de tournées. La recherche automatique d'architectures de réseaux de neurones est une piste de recherche que l'on peut considérer [50]. En outre, nous avons vu dans le cas du CVRP que l'ajout de l'angle au dépôt améliore la convergence du modèle. Il n'est pas exclu que d'autres caractéristiques d'intérêt puissent être trouvées dans de futurs

travaux.

Sur la nature algorithmique, nous abordons les algorithmes d'entraînement et d'exploitation du modèle pour la recherche de meilleures solutions. Bien que la majorité des méthodes se fondent sur l'apprentissage par renforcement pour entraîner les modèles de réseaux de neurones, il n'est pas à écarter que nous puissions obtenir de meilleurs modèles par apprentissage supervisé, particulièrement si ceux-ci sont couplés avec une sélection méticuleuse d'instances d'entraînement. Concernant les algorithmes d'exploitation, notre dernière contribution, ainsi que notre état de l'art, montrent qu'il est intéressant de considérer des approches hybridant apprentissage et (meta)heursitiques. Ceci nous invite à repenser ce qu'il est judicieux de déléguer à un modèle d'apprentissage et qu'il faut traiter par une méthode définie manuellement.

Sur le type de problème traité, nous pouvons distinguer l'étude de problèmes complexes qui intègrent des dimensions difficilement captables par les approches d'optimisation classiques. Par exemple, intégrer l'expérience des chauffeurs à travers un historique de leurs livraisons pour mieux estimer les temps de trajet, ou prévoir la demande des clients sur un horizon temporel pour choisir le moment le plus approprié pour les servir. Nous pouvons également envisager, dans ce cadre, des modèles capables de résoudre une famille plus large de problèmes de tournées, en s'adaptant à la présence ou non de certaines contraintes dans le problème.

Sur l'explicabilité, la nature "boîte noire" des réseaux de neurones rend difficile la compréhension de ce qui est réellement appris à partir des données [75]. De futurs travaux peuvent être initiés afin de mieux comprendre les politiques apprises par les réseaux de neurones profonds. Cet aspect peut être particulièrement crucial si nous voulons appliquer le cadre de l'optimisation combinatoire neuronale à des problèmes de tournées du monde réel. En outre, l'explicabilité peut aider à détecter les biais dans les algorithmes appris. Par exemple, dans le cas du problème de coivoiturage, nous pouvons assurer un service équitable en identifiant tout biais potentiel pouvant conduire à la priorisation ou au traitement préférentiel des quartiers riches par rapport aux autres.

Ces travaux ont fait l'objet des publications suivantes :

- Yaddaden, A., Harispe, S., Vasquez, M., & Buljubašic, M. (2020, June). Apprentissage Automatique pour l'Optimisation Combinatoire: Étude du problème du voyageur de commerce. CNIA 2020 - Conférence Nationale en Intelligence Artificielle.

- Yaddaden, A., Harispe, S., & Vasquez, M. (2021, April). Apprentissage par transfert: du TSP au VRP. In ROADEF 2021-22e congrès annuel de la société Française de Recherche Opérationnelle et d'Aide à la Décision.

- Yaddaden, A., Harispe, S., & Vasquez, M. (2022, February). Évaluation empirique des modèles d'apprentissage profond pour le problème de tournées de véhicules avec contrainte de capacité. In ROADEF 2022-23ème congrès annuel de la Société Française de Recherche Opérationnelle et d'Aide à la Décision.

- Yaddaden, A., Harispe, S., & Vasquez, M. (2022). Is Transfer Learning Helpful for Neural Combinatorial Optimization Applied to Vehicle Routing Problems?. Computing and Informatics, 41(1), 172-190.

- Yaddaden, A., Harispe, S., & Vasquez, M. (2022, July). Neural Order-First Split-Second Algorithm for the Capacitated Vehicle Routing Problem. In International Conference on Optimization and Learning (pp. 168-185). Cham: Springer International Publishing.

- Yaddaden, A., Harispe, S., & Vasquez, M. (2023, February). Une méthode à base d'apprentissage par renforcement pour le problème de tournées de véhicules avec contrainte de capacité. In ROADEF 2023-24ème édition du congrès annuel de la Société Française de Recherche Opérationnelle et d'Aide à la Décision.

Un traitement détaillé de nos travaux est proposé dans le manuscrit rédigé en anglais qui suit.

Remerciements

Ce travail de thèse n'aurait pas été possible sans le soutien et les conseils de nombreuses personnes. Je tiens à exprimer, par ces quelques lignes, mon infinie gratitude.

Tout d'abord, je remercie Michel Vasquez et Sébastien Harispe pour leur encadrement de qualité, leurs précieux conseils et leur patience durant ces années de thèse.

Je tiens aussi à remercier Messieurs El-Ghazali TALBI et Jin-Kao HAO d'avoir bien voulu être rapporteurs pour ma thèse, tout comme je remercie Mesdames Christine SOLNON et Audrey DUPONT, ainsi que Monsieur Rodolphe GIROUDEAU, d'avoir bien voulu faire partie de mon jury de thèse.

Je tiens aussi à exprimer ma gratitude envers le personnel et les enseignants-chercheurs de l'IMT Mines Alès et d'EuroMov DHM pour m'avoir accueilli et permis de réaliser mon travail de thèse. Je remercie également le personnel et les enseignants-chercheurs de Grenoble INP – Génie Industriel et du G-SCOP pour la formidable expérience que représente mon année d'ATER.

Je ne remercierai jamais assez mes parents qui ont tout sacrifié pour leurs enfants. Je vous dois plus que ce que les mots peuvent exprimer.

Je remercie mon frère Khalef, mes sœurs Farida et Rachida, ainsi que Mamie, pour leurs conseils, encouragements et leur aide précieuse.

Finalement, je remercie mes collègues doctorants de l'IMT Mines Alès avec qui j'ai partagé des moments inoubliables. Merci à Jihane, Alexandre, Jérémy, Antoine, Bastien, Mayssa, Quentin, Hamza, Nasir, Sakhi, Yann.

Summary

List of Figures

List of Tables

Glossary

A2C Advantage Actor Critic. 64, 77

AM Attention Model. 66–68, 77, 186

CPU Central Processing Unit. 5

CVRP Capacitated Vehicle Routing Problem. 8, 9, 12, 15–20, 22–26, 47, 55–60, 63–69, 71, 74, 76–79, 82–86, 90, 134, 153, 185, 187, 190

CVRPTW Capacitated Vehicle Routing Problem with Time Windows. 66, 67, 72, 77, 79

DACT Dual-Aspect Collaborative Transformer. 74, 79, 81

DPDP Deep Policy Dynamic Programming. 76, 82

DQN Deep Q-Network. 50, 171–174, 181

DRL Deep Reinforcement Learning. 77

EVRPTW Electric Vehicle Routing Problem with Time Windows. 66, 77

GAT Graph Attention Network. 46, 69, 78, 134–136, 140, 150, 190

GCN Graph Convolutional Network. 46, 68, 72, 76, 78, 134, 135, 150, 190

GLS Guided Local Search. 77

GNN Graph Neural Network. 45–47, 78, 90, 133–135, 137, 140, 150, 151, 188

GPU Graphics Processing Unit. 5

GRU Gated Recurrent Unit. 35, 68, 69, 72, 78, 79, 129, 133, 138

HCVRP Heterogeneous Capacitated Vehicle Routing Problem. 64, 77

HDD Hard Disk Drive. 5

HGS-CVRP Hybrid Genetic Search - CVRP. 26, 58, 76, 82, 190

JAMPR Joint Attention Model for Parallel Route-Construction. 66, 78

KL Kullback-Leibler divergence. 68, 78, 159

LKH3 Lin-Kernighan-Helsgaun heuristic. 58, 59, 74, 76, 190

LNS Large Neighborhood Search. 57, 66, 70–72, 76, 77, 79, 190

LSTM Long Short-Term Memory. 35, 56, 57, 62–64, 76, 77, 79

MCTS Monte Carlo Tree Search. 59, 76, 78, 83

MDAM Multi-Decoder Attention Model. 68, 78, 82

MDP Markov Decision Process. xxix, 48, 51, 60, 61, 70, 100

MDVRP Multi-Depot Vehicle Routing Problem. 69, 78

MLP Multi-Layer Perceptron. 27, 28, 33, 34, 45, 63–65, 68, 71, 73, 74, 76–79, 158–160

NCO Neural Combinatorial Optimization. 60, 62, 63, 65, 66, 76, 77, 101, 128, 129

NLNS Neural Large Neighborhood Search. 71, 79

NOFSS Neural Order-First Split-Second. xxii, xxvi, xxix, xxx, 120, 128–130, 132–134, 141–146, 148–150, 152, 153

OP Orienteering Problem. 65, 77–79

OPTW Orienteering Problem with Time Windows. 66, 77, 84

PCTSP Prize Collecting Traveling Salesman Problem. 14, 65, 77–79

POMO Policy Optimization with Multiple Optima. 66, 78, 83, 84

RAM Random Access Memory. 5

ReLU Rectified Linear Unit. 29, 43, 46, 134, 139, 175–178

RFCS Route-First Cluster-Second algorithm. xxix, 88, 122, 141, 142, 144, 145, 147, 148

RHP Ride-Hailing Problem. 9, 10, 76

RLHF Reinforcement learning from human feedback. 191

RNN Recurrent Neural Network. 35, 37, 38, 41, 43, 77, 90

SDVRP Split Delivery Vehicle Routing Problem. 63, 65, 77–79

SPCTSP Stochastic Prize Collecting Traveling Salesman Problem. 77–79

SSD Solid-State Drive. 5

SVRP Stochastic Vehicle Routing Problem. 77

TL Transfer Learning. 108–110, 112–115

TSP Traveling Salesman Problem. 6, 9, 12–14, 16, 17, 20, 22, 23, 34, 36, 41, 55–62, 65–67, 69, 74, 76–79, 82–85, 190

VRP Vehicle Routing Problem. xxi, xxix, 1, 2, 7–12, 15–22, 24, 25, 34, 38, 43, 53, 55, 57, 59–61, 63–65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 90, 93, 99, 120, 126, 129, 134, 155, 183

Introduction

Road transport: key sector and challenges

Road transportation of people and goods is an essential activity in modern society. It facilitates economic and social transactions by allowing people to travel, work, study, or access healthcare. Similarly, the transport of goods stimulates the economy while allowing the distribution of vital goods such as food, clothing, medicines, etc. As an illustration, in France in 2017, the road transportation of goods represents 89% of the inland traffic, with a turnover of 43 billion euros, while road passenger transport represents a turnover of 7.1 billion euros [11].

In the context of smart cities and the industry 4.0, urban logistics must be rethought with all the challenges that this imposes in terms of reducing noise, congestion, pollution and increasing the quality of services and customer satisfaction. To meet these challenges, smart transportation and logistics propose to rethink both the modeling of routing problems and the algorithms used to solve them [92]. Indeed, when designing delivery plans, decision makers often model vehicle routing problems (VRPs) to control costs and/or maximize profits. Furthermore, with the advent of the Internet of Things and smartphones, the amount of data that businesses have access to has increased exponentially; the transport sector is no exception [204]. This has created new opportunities, such as offering customers the ability to track in real-time their orders and to consider their preferences in terms of delivery schedules. Delivery plans must then accommodate order cancellation or rescheduling. In some cases, very short decision times are imposed on

the algorithms to deliver a new plan.

In the light of all this, it appears that it is important to study two types of VRPs: deterministic vehicle routing problems where all the problem data are known in advance, and problems with stochastic or dynamic characteristics where the problem data change or appear over time [146]. The first type is essential when it is not necessary to plan in real time, such as when planning supermarket deliveries by suppliers. As for the stochastic/dynamic problems, they are essential to consider uncertainties related to customer demands, service requests and travel times [184]. Tackling these problems opens perspectives in designing new algorithms that will, in the long term, contribute to advances in the supply chain optimization, in general.

Algorithmic challenges of transport problems

Combinatorial optimization seems to be an inherent part of urban logistics, as many of the problems encountered can be modeled as discrete-variable decision problems. Optimization allows us first to quantify the proposed solutions to the problems and, secondly, to look for better ones. Vehicle routing problems are one of the most frequently encountered classes of problems that are optimized daily [184]. These problems are known to be $\mathcal{NP}-$hard, meaning that, until today, there is no deterministic polynomial time algorithm that optimally solves all the instances of these problems [141].

The research for efficient algorithms for solving combinatorial optimization problems has resulted in a plethora of efficient solution methods, ranging from exact methods to (meta)heuristic methods [141, 173]. However, challenges remain, especially regarding the quality of the solutions and the execution time. Often, heuristic methods are favored for their reasonable ratio in terms of solution quality and execution time. Different handcrafted heuristic strategies, that consider the problem characteristics, have been designed. These handcrafted heuristics generally require expert knowledge on the problem and have variable performance depending on the instance that is tackled. Overall, when tackling a new problem with different constraints, heuristics have to be adapted and tested to select the one that performs better. This can be a repetitive, tedious and time-consuming

task for the Operations Research engineers [27].

In this context, a highly sought-after search method would be a method (1) built automatically by the computer, (2) capable of producing good quality solutions in reasonable times, and (3) that can quickly adapt to changes in the constraints of the problem. From this perspective, Artificial Intelligence, and more particularly Machine Learning, seem to be promising fields that open perspectives in the automatic design of algorithms capable of discovering solution search strategies. In this thesis, we aim at studying the vehicle routing problems from this perspective.

Machine Learning: an Artificial Intelligence approach

At the intersection between Computer Science, Statistics, Neurosciences and Optimization, Machine Learning is an Artificial Intelligence subfield that can be viewed *"as searching through a large space of candidate programs, guided by training experience, to find a program that optimizes the performance metric."*[87]
More formally, Mitchell defines Machine Learning as follows [128]:

Definition. A computer program is said to learn from experience E regarding some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.

In this definition, the experience refers to the gathered data used for training, the task is the end goal we want the program to achieve and the performance measure gives us feedback on the quality of the learned program. For example, let us consider a handwriting recognition problem where a computer program has to distinguish the ten different digits from zero to nine (the famous MNIST task [110]). The data that we gather is a set of different handwritten digits with their corresponding digit value (label). In this case, the task is to correctly classify any handwritten digit to its corresponding digit. A simple measure of performance for the learned program is to count the number of correctly and wrongly classified samples not used during the learning phase. We would consider learning to be more or less successful based on the evaluation of the performance measure.

Generally, we can classify machine learning algorithms into three main paradigms [84]:

- Supervised learning: the most encountered type of machine learning algorithms, today. It considers learning from labeled datasets. The previous example corresponds to that setting.

- Unsupervised learning: in this case, we consider unlabeled datasets. The goal is generally to find patterns and the hidden structures in datasets. An example of this type is data clustering.

- Reinforcement learning: this type of machine learning uses unlabeled datasets. In reinforcement learning, an agent interacts with an environment by taking actions that are determined by the observation of the current state. These actions generate numerical values known as rewards. The objective of reinforcement learning is to find an optimal behavior policy for the agent that maximizes the long-term reward. An example of this type is learning a program that successfully plays chess. The feedback in this case could corresponds to +1 if the program wins, 0 if it's a tie, and -1 if it loses.

We can also distinguish other types of learning such as semi-supervised learning where part of the dataset has labels, self-supervised learning that is considered sometimes as a subfield of unsupervised learning and aims at finding patterns in data by creating an artificial supervised task, few-shot learning that considers learning from few data or transfer learning that uses knowledge gained from solving one task to tackle a different but related task.

Traditional problem-solving approaches propose to tackle problems by first studying, then formulating explicit rules and a set of instructions that the computer must follow to produce a solution. Depending on the problem to tackle, this can be a very complex and difficult task to do. For example, the problem of finding the shortest path in a road network has seen the development of many algorithms with different handcrafted rules, such as Bellman's algorithm [19], Floyd-Warshall algorithm [190] and Dijkstra's algorithm [43].

Machine learning proposes a different paradigm (see Figure 2). Data is gathered according to the task to solve. Instead of handcrafted rules, an end goal is presented to the computer. Depending on the type of machine learning, this can be, for instance, the labels of the training samples in the case of supervised learning,

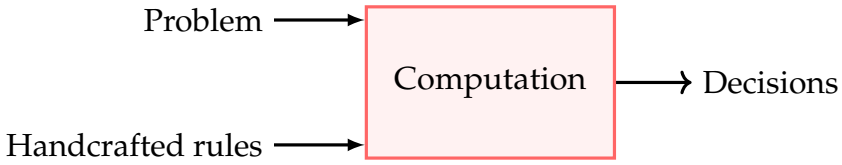


Figure 1: The traditional problem-solving approach.

or a score function that evaluates the performance of the algorithm in the case of reinforcement learning. The output is the computer program that can tackle unseen instances of the same problem.

Figure 2: Machine learning problem-solving approach.

The last decades have seen the successful application of machine learning in several domains, ranging from facial recognition [13], automatic text translation [31], intelligent conversational agents such as chatGPT [137], to the creation of intelligent agents for combinatorial games such as chess or go [166]. This is mainly due to recent advances in computer hardware for (1) computation, with more powerful CPUs and GPUs, (2) data collection using sensors (e.g., IoT devices), social networks, etc. (3) data storage such as SSDs. The low cost of computer hardware such as RAMs or HDD has allowed the development of cloud computing, which has democratized access to computing and storage resources. This has greatly facilitated the emergence of several application cases that contributed to the success of the domain. Moreover, government initiatives such as Grid5000[1] or Jean Zay[2] supercomputers are accelerating research and algorithm developments [23]. Furthermore, several theoretical advances have been made, whether in optimization (learning can most often be framed as solving a complex non-convex optimization problem) [96], the introduction of new machine learning models (e.g., deep neural

[1] `https://www.grid5000.fr/w/Grid5000:Home`
[2] `http://www.idris.fr/eng/jean-zay/jean-zay-presentation-eng.html`

networks [109]), or a more in-depth understanding of the mechanisms that stabilize the models training (e.g., regularization[3][66]). Moreover, it should be noted that several machine learning development frameworks, such as Pytorch [143], have recently been introduced, thus simplifying the design of new models. Finally, it is worth mentioning that several platforms now provide access to databases of pre-trained models as well as datasets, such as those available on HuggingFace[4].

The idea of using machine learning models in the field of combinatorial optimization is not new. In the earlier studies, the utilization of various artificial neural network models were benchmarked on the Traveling Salesman Problem (TSP)[168]. These methods have been abandoned because they have not been successful on optimization problems, especially when compared to metaheuristics [168]. But due to the advances that we mentioned above, there is a renewed interest in using machine learning for optimization problems. Especially, over the past seven years, there has been a growing interest in techniques relying on deep neural networks and deep reinforcement learning, leading to the emergence of the Neural Combinatorial Optimization framework. These methods aim to revive the pioneering works on the use of neural networks to learn search strategies for combinatorial optimization problems. Neural networks are particularly interesting as they are universal approximators, meaning that they can approximate a wide variety of continuous functions in $\mathbb{R}^n$ [78]. The reformulation of a combinatorial optimization problem from a discrete space to a continuous space would thus allow us to transform the problem into the search for a continuous function approximated by a deep neural network.

This thesis presents an exploratory investigation into the feasibility of using deep neural networks to tackle difficult vehicle routing problems. Therefore, it should be noted that the observed performances of these approaches, while promising, do not yet surpass those of established state-of-the-art methods. In the following section, we present the research questions studied in our work.

[3]Regularization is a set of techniques used to prevent the model from learning the training dataset by heart. Indeed, the goal of the program learned via machine learning is to be able to handle examples of the same problem given during training.

[4]https://huggingface.co/

Research questions and thesis objectives

The emerging field of machine learning for combinatorial optimization problems represents a considerable opportunity for the development of more efficient algorithms to tackle a wide range of problems. It is also a source of scientific questions that invite to propose contributions in very diverse aspects. For example, one can question which learning paradigm to use? Which model is suitable? Can a learned strategy alone compete with handcrafted heuristics, or is it better to hybridize both?

The aim of this thesis is to contribute in some research directions regarding the use of deep neural network models for solving vehicle routing problems. We focus on two different classes of vehicle routing problems: deterministic and stochastic versions. We identified the following research questions:

- Is it possible for a deep neural network to achieve performance comparable to or better than that of handcrafted heuristics developed by domain experts?
- What are the performances of deep neural network-based approaches on known benchmark datasets under different training settings?
- Do deep neural networks still fall short of (meta)heuristics in terms of solution quality and execution time when used to solve deterministic VRPs?
- Would it be possible to reuse a deep neural network trained on a specific problem to tackle a similar problem?
- Can we use a deep neural network to learn an implicit search strategy for the VRP?
- What are the performances of deep neural networks when used to solve a dynamic VRP?

Thesis outline

In this thesis, we study the VRP from the deep learning and reinforcement learning perspective. In particular, we are interested in the contribution of these two areas

of machine learning in both deterministic and dynamic/stochastic VRPs. In the first case, the Capacitated VRP is studied. In this variant, we have a set of customers and a homogeneous fleet of vehicles. The goal is to minimize the delivery cost of the customers (estimated by the total distance traveled) without exceeding the capacity of the vehicles. As complete information is available regarding the clients, vehicles, and road network, this serves as an ideal case study for evaluating novel methodologies. Uchoa et al. [175] wrote about the problem: *"Being the most basic variant, it is a natural test bed for trying new ideas. Its relative simplicity allows cleaner descriptions and implementations, without the additional conceptual burden necessary to handle more complex variants. Successful ideas for the CVRP are often later extended to more complex variants"*. In the second case, the Ride-Hailing Problem is studied. It is a problem where a central operator handles a fleet of vehicles to serve requests that arrive during the time of service. There is a strong stochasticity due to the travel time depending on the zones where the requests are located, to simulate the traffic congestion. The goal is to maximize the total gain to the ride-sharing company by determining which customers to serve and which vehicle to assign to the customers.

The diagram of Figure 3 illustrates the structure of the manuscript.

In Chapter 1, we give a brief introduction to the VRP. We review the main deterministic and dynamic variants. A review of solving methods is presented to situate our work in relation to the existing approaches. Then, we introduce the key concepts in machine learning that are necessary to understand the work of this thesis, namely deep neural networks and reinforcement learning. In the second part of the chapter, we present the state of the art of different deep learning and reinforcement learning methods for solving VRPs. We propose a taxonomy of the different contributions of the state of the art according to different criteria such as the learning methods and the type of neural network used. We also discuss the different search methods used to exploit the strategies learned by neural networks to find better quality solutions. We then present the training and evaluation protocols implemented for the deep neural network-based solution approaches, as well as initial empirical evaluations on recognized instances in the literature of the VRP to better situate their performance.

Chapter 2 presents our first contribution, which deals with the study of trans-

fer learning on VRP. Our study is based on the TSP and the CVRP. It aims to evaluate if a deep neural network model trained to solve one of these problems can be used to solve the other problem with a few additional training steps. Extensive computational experiments show that when relatively little training data is available, transfer learning allows our model to learn faster than if it was trained from scratch.

Chapter 3 presents NOFSS (Neural order-first split-second), a two-steps method based on the order-first split-second paradigm used to tackle the CVRP. It hybridizes neural networks with a shortest path algorithm. We propose indirectly to learn a resolution strategy via the exploration of the space of giant tours. We conduct extensive experiments using various deep neural network architectures and compare our results to classical heuristics for CVRP. Our results show that our method is competitive with the state of the art in terms of result and execution time. In addition, we conduct experiments on CVRPLib based on the evaluation protocol introduced in Chapter 1.

In Chapter 4, we study a stochastic and dynamic variant of the vehicle routing problem: the Ride-Hailing Problem (RHP). We formulate it as a reinforcement learning problem, and we propose to explore various solution algorithms. We conduct a comprehensive set of experiments to demonstrate the effectiveness of learning-based methods for addressing the problem.

Finally, we conclude by summarizing our contributions and discussing potential future research directions.

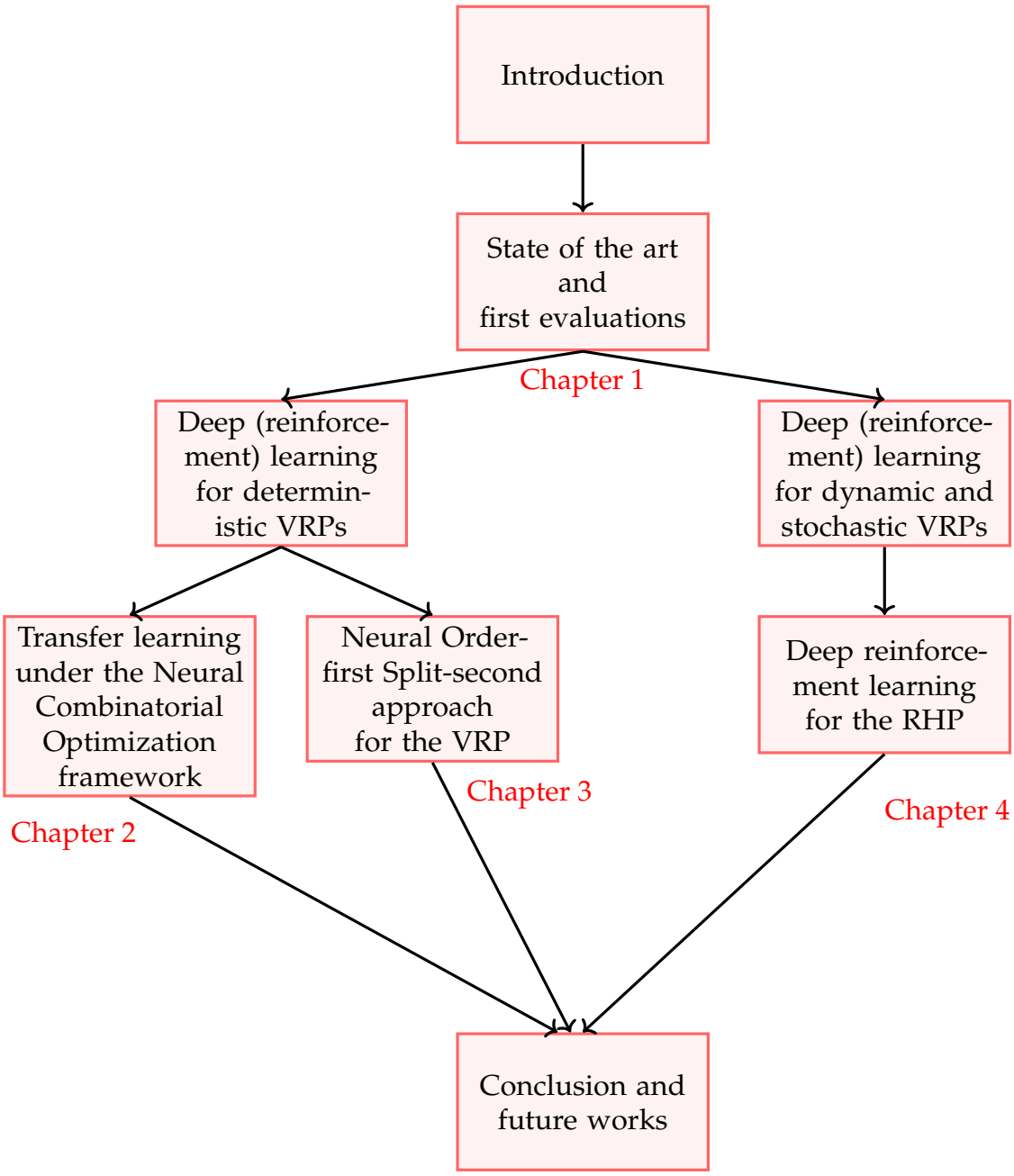


Figure 3: Organization of the manuscript.

Chapter 1

State-of-the-art

As a subfield of mathematical optimization, combinatorial optimization deals with finding an optimal object from a finite set of objects [141]. Typically, combinatorial optimization problems occur when we are seeking for the best configuration among the feasible ones. The combinatorial nature of the problems makes the exhaustive search of solution intractable and would result in a *combinatorial explosion* in terms of running time and/or memory. These kinds of problems are ubiquitous in industry, logistics or healthcare, to name a few. They also bear interest in complexity theory and algorithmic theory, since they are a source of many advances in computer science.

This chapter focuses on introducing vehicle routing problems (VRPs) as a class of difficult combinatorial optimization problems. We review the main VRP variants as well as the key solution methods used to tackle them. Furthermore, we introduce the main machine learning concepts that we will use throughout this thesis, mainly the type of artificial neural networks that are widely used, and the reinforcement learning algorithms used to train them. Then, we review the end-to-end learning-based methods for the VRP. We will focus on the principal VRP variants as introduced in the first section. In this review, we will distinguish the different axes under which end-to-end learning for VRPs has been addressed (i.e., learning method, type of neural network, learning construction or improvement strategy, etc.). The different search strategies used to exploit the learned model are also exposed. Furthermore, to better understand the potential and limitations

of the models, we conduct empirical evaluations of these approaches on the Capacitated Vehicle Routing Problem (CVRP), as a key representative VRP under various scenarios.

1.1 Overview of the vehicle routing problems

1.1.1 The traveling salesman problem

The traveling salesman problem (TSP) is one of the well-known and most studied combinatorial optimization problems. The problem can be formulated as follows: *Given a set of n cities and the distances between each pair of cities, what is the shortest possible path that traverses all the cities once, and ends at the beginning city?*

We can model this problem using graph theory. A TSP can be represented with a graph $G = (V, E)$ with the vertices being the set of n cities to visit, and the edges of the graph being the routes that connect the cities. Each edge has an associated weight, being the distance between the cities. Generally, the TSP refers to the Euclidean TSP, i.e., Euclidean distances are used as a cost between two cities. Therefore, solving the TSP implies finding the shortest Hamiltonian cycle in the graph. Figure 1.1-(a, b)[1] shows an example of an instance of 32 cities and its corresponding solution[2], the red square node being the first visited city. A candidate solution of the TSP can be seen as a permutation of the order of visiting cities. Thus, for an instance of n cities, we would have to search among $n!$ candidate solutions. Therefore, an exhaustive search is intractable for large values of n.

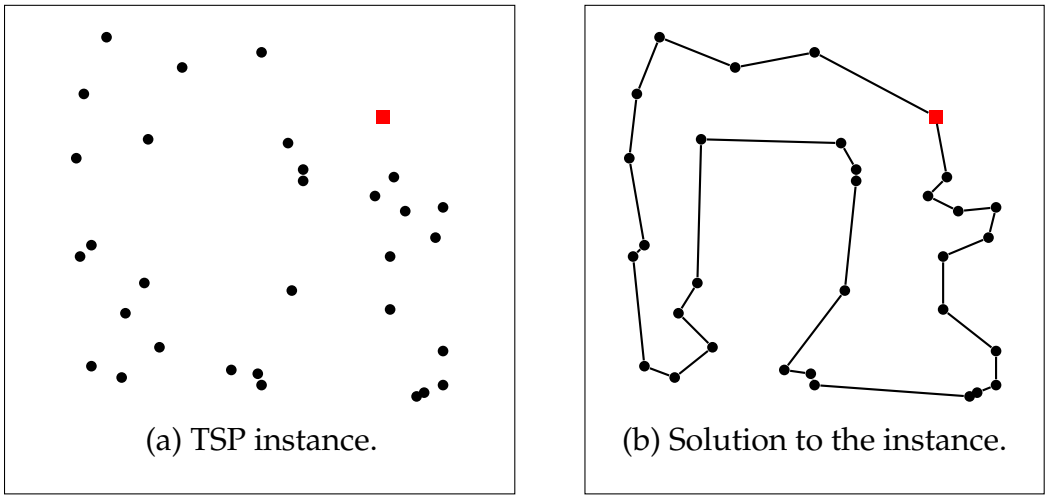


Figure 1.1: Example of a TSP instance.

The first work on the TSP dates back to the 1800s [160], but it took until 1954 with the seminal work of Dantzig, Fulkerson and Johnson to have a description of a method based on linear programming to solve TSP instances [40]. They found an optimal tour for a 49 cities instance, which was considered as large scale at that time. Since then, many advances were made to tackle the problem. The size

[1]The graph considered is a complete graph, i.e., we have a connection between each pair of vertices, but we omit them in the figure for readability.

[2]The instance is the `A-n32-k5` from Augerat et al. [9] benchmark.

of the instances used to benchmark the solution methods continued to grow over the decades, from a few hundred cities to billions of cities. World TSP, which has 1.9 million cities, is one of the large-scale benchmarks that challenges the TSP community to test their algorithm on [38]. The best known solution given by LKH heuristic is at most 0.047% greater than the optimal tour [38]. In the present day, the largest TSP instance is the Galaxy TSP, with 1.69 billion nodes constructed from the catalog of stars in the Milky Way [45]. Of course, there exist small to middle scale benchmark instances that were made available for researchers to test their algorithms. These instances are compiled in the Traveling Salesman Problem Library (TSPLIB) from various sources and with various properties [153]. Furthermore, researchers often rely on Concorde as a benchmark for comparing the results of their own TSP algorithms due to its reputation as one of the best TSP solvers available [6].

Although significant advances have been achieved in tackling this problem, it remains that it is a $\mathcal{NP}-$hard problem, which means that until today, no algorithm is known to optimally solve all the instances of the problem in polynomial time [141]. Thus, the TSP remains one of the most important and representative combinatorial optimization problems. It has led to many algorithmic advances as it serves as a test-bed for numerous algorithms such as simulated annealing, ant colony optimization, or iterated local search to name a few [98, 37, 117]. As we will see throughout this chapter, the methods we are going to present are no exception to the rule, as many of them were first tested on this problem.

It is relevant to mention that as real-world applications and needs have emerged and evolved, variants of the TSP have been proposed and studied such as the multiple TSP (mTSP) in which multiple salesmen are employed to visit a set of cities [18], the TSP with Time Windows (TSPTW) where cities can be visited only within their time windows [47], the Prize Collecting TSP (PCTSP) in which a prize and a penalty are associated with each city. In the latter problem, the salesman must visit a subset of the cities to collect a certain amount of prizes while minimizing the total traveled distance and additional penalties of the unvisited cities [14].

1.1.2 The vehicle routing problem

First introduced in the pioneering work of Dantzig and Ramser in 1959 as the *"truck dispatching problem"* for the delivery of gasoline to gas stations [41], the VRP concerns the delivery and/or the collect of goods between depots and clients, as well as the transport of persons from one place to another. Defined as it is, we can make the observation that there are multiple scenarios that can fit into it. To be more rigorous, we will introduce the problem through the most studied variant, namely the Capacitated Vehicle Routing Problem (CVRP).

In the CVRP, we have a set $N = \{1, \ldots, n\}$ of n clients with demands $q_i > 0$, $i \in N$, a single depot, denoted as 0, with a fleet of homogeneous vehicles with a loading capacity Q, and an associated travel cost c_{ij} between a pair of locations $(i, j) \in (N \cup \{0\})^2$. For convenience, we associate a zero demand to the depot, i.e. $q_0 = 0$. The goal is to find a set of routes, such as the total travel cost is minimal and such that the following constraints are satisfied:

1. A route begins at the depot, traverses a set of clients and ends at the depot;
2. The total demand of a set of clients in a route does not exceed the vehicle's capacity;
3. A client belongs to one and only one route;
4. All clients must be assigned to a route;
5. When a client is visited, their request must be fulfilled completely.

The demands are assumed to be less than or equal to the vehicle's capacity, i.e, $q_i \leq Q$, $\forall i \in N$. Furthermore, we consider symmetric travel costs in the form of the Euclidean distance between the clients and the depot, i.e., $c_{ij} = c_{ji}$, $(i, j) \in (N \cup \{0\})^2$.

Solving the CVRP involves two kinds of decisions: (1) grouping the clients into clusters that respect the vehicle capacity, and (2) sequencing the clients to get routes with minimum travel distance [150]. The CVRP therefore contains two problems: a bin-packing problem to define the clusters and a traveling salesman problem for the visit order of each cluster, starting from and ending at the depot.

Let us note that the two tasks are interdependent, i.e., the cost of clustering depends on the order of visit and the cost of the order of visit, depends on the clusters. There is not, *a priori*, an order in the operations to favor over another.

Figure 1.2 depicts an example of a VRP solution by taking the previous TSP instance as an example (Figure 1.1). The red square is the depot, and the black circles are the clients.

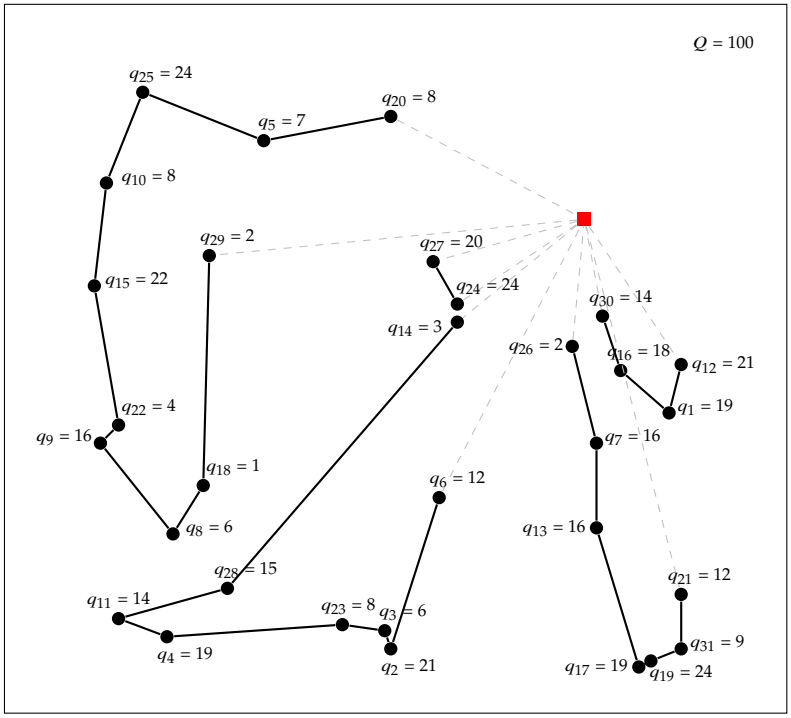


Figure 1.2: Example of a solution to a CVRP instance with 5 tours.

The bin-packing problem associated with CVRP allows us to give a lower bound on the number of vehicles (or routes), k, needed to serve customers. This is given by the following formula:

$$k = \left\lceil \frac{\sum_{i=i}^{n} q_i}{Q} \right\rceil$$

Let us observe that this is only a lower bound, i.e., the number of vehicles necessary to solve a CVRP instance may be greater than this bound[3]. An obvious upper

[3]For example, consider the CVRP problem where the capacity $Q = 5$, and there are 3 clients with equal demands of 3 units ($q_1 = q_2 = q_3 = 3$). Based on the lower bound, $k = \left\lceil \frac{3+3+3}{5} \right\rceil = 2$.

bound of the number of the number of vehicles is obtained by assuming that each client is assigned to a single route, i.e., $k \leq n$.

The problem admits a formulation based on graph theory. We consider $G = (V, E)$ an undirected graph with $V = \{0\} \cup N$ being the set of vertices representing the depot and the clients, and the edge set $E = V \times V$ corresponding to the road network that connects the clients and the depot. The graph G is assumed to be complete. We assign weights to the vertices and edges corresponding respectively to the client demands q_i, $i \in V$ and the travel costs c_{ij}, $(i, j) \in E$. The CVRP consists of finding a set of simple circuits with the minimum total cost and such that the above constraints are respected.

Like the TSP, this problem is $\mathcal{NP}-$hard [112], although from a practical perspective, the VRP is much more difficult to solve than the TSP for instances of the same size [106].

Let us note that if we consider a sufficiently large vehicle load capacity such that $\sum_{i=1}^{n} q_i \ll Q$, we can formulate either a multiple traveling salesman problem (mTSP), if we consider using the whole fleet of k vehicles, or a TSP if we consider a single vehicle. Thus, we consider that the CVRP generalizes the mTSP and the TSP. Similarly, adding constraints to CVRP gives us a variant that generalizes the problem. We will detail the variants in the following section.

1.1.3 VRP variants

As pointed in the previous section, the CVRP is the base variant of the VRP. However, to model more realistic routing problems, additional constraints have been added, depending on the case study. We can classify the different variants according to different criteria following the types of vehicles, customers, depots and road network. The variants may also depend on the type of objective function. It is possible to consider a maximization problem, where the total profit is assumed to be a quantity of interest, or a minimization problem where a cost function is to be minimized. It is also possible to define multi-objective VRPs, where we have more than one objective function to optimize. While there is abundant

However, since a visited client must have its request fully satisfied, they can only be delivered in separate routes, which means that $k = 3$.

literature about this topic, we consider these types of problem out of the scope of this thesis. Readers may refer to recent surveys on this particular subject for additional information [91, 203].

For the clients, there may be several constraints related to real-world applications. One of the most common being the periods of the day in which the clients can be served, referred to as *time windows*. In addition, waiting times may be observed when arriving at a client's location. This reflects the time required to load or unload the goods. The base variant is termed the Vehicle Routing Problem with Time Windows [39]. Other clients characteristics include the type of request that can be picked up and/or delivery request. In addition, deliveries may be split, which allows a vehicle to partially fulfill a demand when visiting a client.

Concerning the vehicle characteristics, the fleet can be made of vehicles with different loading capacities, it is then considered as heterogeneous. The vehicles can also be of different type, for example electric vehicles, or a mix between electric and thermal vehicles. Recently, multimodality has become a topic of interest for VRPs, for reducing the travel costs and congestion. For example, cooperations between trucks and scouts, or between trucks and drones are increasingly considered.

In the CVRP, we considered only one depot, but the problem can be generalized to multi-depot. Besides this, we can also find satellite depots, in the two-echelon VRP, where intermediate delivery depots are considered. The constraint of returning to the depot can also be omitted, thus formulating an Open VRP.

Depending on the characteristics considered, one can define numerous VRPs. We can combine the above-mentioned properties to build an exponentially huge number of variants. For example, we can combine time windows with a heterogeneous fleet VRP to solve a heterogeneous fleet VRP with time windows. Furthermore, depending on data availability and problem uncertainties, we can define static or dynamic variants as well as deterministic or stochastic ones. Table 1.1 summarizes the different types of data availability and uncertainty associated with problem characteristics [146]. We can define: (i) *static deterministic* variants where all problem inputs are known before the beginning of the resolution process (e.g., CVRP), (ii) *static stochastic* variants where stochasticity is associated to some

problem characteristics, such as stochastic demands (VRPSD), (iii) *Dynamic deterministic* variants where part of problem inputs are revealed on the fly, and (iv) *Dynamic stochastic* variants where problem inputs are revealed while performing the resolution and some inputs involve stochasticity. A common example of this type is the dial-a-ride problem, where customers' requests in an ongoing fashion and stochastic travel times are associated with vehicles [158].

		Uncertainty	
		Deterministic	Stochastic
Data availability	Static	Static deterministic	Static stochastic
	Dynamic	Dynamic deterministic	Dynamic stochastic

Table 1.1: Types of VRPs based on data availability and uncertainty.

1.1.4 Solution methods

In the 63 years of their existence, a plethora of solution methods have been developed for the different variants of the vehicle routing problems. In the context of this thesis, we distinguish between handcrafted methods, learning-based ones, and hybrids between these two. In handcrafted algorithms, one has to describe the solution method used to tackle the problem. They generally require expert knowledge about the problem. Learning-based methods, on the other hand, aim to uncover solutions by not explicitly defining the strategies that generate high-quality solutions. These methods do not require intensive knowledge about the problem. Finally, hybrid methods propose to take advantage of both approaches either by incorporating a method of one type inside another, or by running a method of one type before another. In what follows, we review the handcrafted methods for the CVRP. The other solution methods involving learning, which are at the center of this thesis, will be reviewed in their specific section.

1.1.4.1 Exact solution methods

Branch and bound [105]. This algorithm uses a divide and conquer strategy on the solution space S and performs two operations:

- Branching: the currently visited solution space S^c is partitioned into two or more, ideally disjoint, subspaces $S_1^c, S_2^c, \ldots, S_m^c$, with $S^c = \bigcup_{i=1}^{m} S_i^c$. This partitioning can be viewed as a generation of new leaves in a search tree. The generated subspaces are stored in a queue. Depending on how the queue is managed, i.e., how a new subspace is selected, the tree can be traversed in Depth-First Search, Best-First Search, Breadth-First Search.

- Bounding: in this phase, a lower bound $LB(S^c)$ of the solution space S^c is computed: $LB(S^c) \leq f(x), \forall x \in S^c$ with $f(\cdot)$ being the objective function associated with the problem. The lower bound has not to be associated with a valid solution in S^c.

Let M denote the objective value of the incumbent solution (the objective value of the best solution found so far). This incumbent value provides, also, an upper bound of the optimal objective value, in a minimization problem. The incumbent solution is updated each time a new solution $x \in S^c$, with $f(x) < M$, is found. The branch and bound algorithm prunes a solution space S^c if $LB(S^c) > M$, otherwise, branching occurs. The algorithm terminates when the subspaces queue is empty.

The first branch and bound used bounds based on the *Shortest Spanning Tree*[4], which is an extension to the CVRP of the 1-tree method introduced for the TSP and a branching scheme based on arcs, i.e., create two solution subspaces where in the first the arc (i, j) is selected to extend the path and another where it is not selected [35]. Fisher extended the 1-tree approach by introducing the $k-$trees approach with a mixed branching scheme based on the arcs and the clients [56]. Other work considers a lower bound based on the b-matching problem and Lagrangian relaxation [127].

Branch and cut. This algorithm is a branch and bound with cutting planes for (mixed) integer programs. The first approach of this type for the VRP proposes a two-index integer program formulation of the problem. The integrality constraints of the variables are relaxed, resulting in a linear program that is solved using the simplex algorithm. If the solution found by the simplex algorithm is feasible for the CVRP, then it is optimal, and the algorithm stops. If it is not feasible, or it

[4]the problem of finding a minimum-cost subset of edges connecting all vertices of the graph.

is greater than the incumbent, then the sub-problem is pruned. Valid inequalities (cuts) are added. When it is not possible to add further cuts, branching occurs on a fractional variable, creating two sub-problems [107]. Many valid inequalities were introduced for the VRP [7, 2, 3, 120].

Column Generation The VRP admits a set partitioning formulation [15]. In this formulation, the set of all feasible routes is considered. Thus, the goal is to select among those routes, the optimal subset. There is an exponential number of these routes. Thus, we resort to column generation. The method starts from a small subset of routes and solves a linear relaxation of the reduced model (the model with only the subset of routes) to get the optimal dual variables associated with the constraints. The dual information is then used to select a new route that will be added to the subset. The algorithm iterates again with the new subset, until no new route can be added. Numerous works adapted the column generation to the VRP, readers may refer to the works in [167, 5, 42]. Readers may also refer to a detailed tutorial of D. Feillet on column generation for the VRP for additional information [53].

Branch-and-cut-and-price This algorithm combines cuts and column generation into a branch and bound. The principle of pricing is to find columns (e.g., sub-routes) with reduced negative costs. This approach is the best performing of the exact approaches for VRPs [59, 74]. VRPSolver, a branch-and-cut-and-price exact algorithm, achieves state-of-the-art results on different VRP variants and is freely available for research [145][5]. Different branch-and-cut-and-price were introduced in the literature, readers may refer to the book chapter by Poggi and Uchoa for a detailed review [147].

1.1.4.2 Heuristic solution methods

Exact solution methods are efficient for small size VRP instances because they find the optimal solution in a reasonable computation time. However, as the size of the instances grows, finding the optimal solution becomes intractable. Heuristic search

[5] Available at `https://vrpsolver.math.u-bordeaux.fr/`

methods are thus used to compute a solution. We can distinguish two types of heuristics for the VRP: constructive heuristics that build a solution from scratch, and improvement heuristics that start from an initial solution and try to find a better one.

Constructive heuristics are, themselves, divided into two categories: single-step methods and two-steps methods.

Single-step methods iteratively build the solution by selecting the next client to visit, via a greedy decision rule. The nearest neighbor is a simple greedy heuristic that can be used to generate a feasible vehicle routing solution. It starts from the depot, and iteratively adds the nearest unvisited client, in terms of Euclidean distance, to the route, until no further client can be added due to capacity constraint. A new route is then started. The algorithm loops until all clients are visited. This heuristic is simple and quick to execute, at the expense of the quality of the solution. Another single-step, that is widely used, is the Clarke and Wright savings heuristic [36]. The heuristic merges routes of the form $(0, \ldots, i, 0)$ and $(0, j, \ldots, 0)$ by applying the savings criterion $s_{ij} = c_{i0} + c_{0j} - c_{ij}, \forall i, j \in N$. The merge with the largest savings is selected at each iteration, until it is not possible to perform the operation. Several improvements were proposed to this heuristic, for example, a parameter λ was added to the savings formula: $s_{ij} = c_{i0} + c_{0j} - \lambda c_{ij}$ [138].

Two-steps approaches for solving the CVRP involve (i) partitioning the clients into feasible clusters regarding vehicle capacity and (ii) ordering them into routes of minimum length. Based on how the two operations are orchestrated, we can distinguish two types of two-steps algorithms: Cluster-first Route-second and Order-first Split-Second.

Cluster-first Route-second. In this type of algorithm, the clients are first grouped together following the vehicle capacity constraint, then a traveling salesman problem is solved for each cluster using an exact solver or heuristics. The Sweep algorithm is the most common algorithm of this type [64]. Feasible clusters are constructed by considering the polar angle between the clients and the depot, then for each cluster a TSP is solved. An extension of this algorithm called the petal algorithm considers generating several routes and selects the final routes of the solution by solving a set partitioning problem [157]. Another work considers obtaining the clusters by solving a generalized assignment problem [57].

Order-first Split-second. These approaches consider first ordering the customers into a sequence called a giant tour, to then, decompose it into a set of feasible tours considering the vehicle capacity. Traveling salesman problem heuristics are used to get a giant tour, and the CVRP tours are optimally extracted from it by solving a shortest path problem. The first documented approach of this type generates the giant tour by random permutation of clients' visit order, followed by a 2-OPT improvement, and then builds the routes using Floyd's algorithm [17]. This type of approach is widely used inside metaheuristics, as it is simple and fast to generate a giant tour. We will detail the approach in Chapter 2 of this thesis, as we will develop a learning-based approach that uses this solution representation for the CVRP.

Starting from an initial candidate solution, improvement heuristics aim at finding a better candidate solution by performing a move from one candidate solution to a neighboring one. We can distinguish inter-route and intra-route improvement heuristics.

Intra-route heuristics correspond to the improvement moves used in TSP. They are applied to one route at a time, and change the order in which clients are visited. Typical improvement heuristics include λ-OPT moves in which λ edges are removed and are replaced by λ other edges. The most used ones being 2-OPT and 3-OPT [115]. Other heuristics include RELOCATE, which changes the position of a client in the tour to another and SWAP, which exchanges the position of two clients in a route.

Inter-route heuristics are moves that are used to change the clients' positions from one route to another. They are adapted from the intra-route heuristics. We can cite the $2-\text{OPT}^*$ which changes two edges from two different routes by two other edges [148]. The RELOCATE can be adapted to remove a client from a route, and insert it into another route. The SWAP move exchanges two clients from two different routes.

1.1.4.3 Metaheuristics

Metaheuristics are high-level heuristic algorithms designed to guide the search procedure for a good quality solution. Generally, they are classified as Local

Search algorithms and Population-based algorithms. In what follows, we briefly review both of the types of metaheuristics applied to the VRP.

Local search algorithms. These methods explore the solution space by moving from one solution to another by exploring its neighbrohood. A recent review of papers published between 2009 and 2015 shows that almost 64% of the metaheuristics used to tackle VRPs were local search algorithms. Among them, 30% are based on Tabu Search, 23% on Variable Neighborhood Search, 15% on Large Neighborhood Search, 12% on Simulated Annealing, 10% on Iterated Local Search, 9% on Greedy Randomized Adaptive Procedure, and 1% on Guided Local Search [49].

Tabu Search. This algorithm explores the neighborhood of a solution and forbids some neighboring solutions that share characteristics with the current one, to avoid cycling. Furthermore, to prevent the local search to become stuck in local optima, solutions with worse cost can be accepted. TABUROUT was among the first tabu search algorithms for the CVRP. The algorithm allows infeasible candidate solutions, and introduces a penalty term to penalize capacity excess [62].

Variable Neighborhood Search. This method uses several neighborhood operators such as 2-OPT, 3-OPT, SWAP, etc. to improve the current solution. Starting from an initial solution, the operators are applied sequentially, one after another. When the last operator is applied, the algorithm restarts again from the first operator. This method has been successfully applied for the CVRP in [104, 24].

Large Neighborhood Search. This metaheuristic alternates between ruin (destroy) and recreate (repair) operations. Starting from an initial solution, the ruin operator removes clients from tours, and the recreate operator inserts them back in other positions so that a new better solution is obtained. A simple but effective method based on string removal and a greedy insertion procedure achieves results competitive with the best metaheuristics [33].

Simulated Annealing [98]. This algorithm takes inspiration from annealing in metallurgy, where heating and controlled cooling is used on materials to make them more ductile. In the algorithm, solutions with worse objective function value than the incumbent are accepted with a certain probability that is a function of the cost of the solution considered and the incumbent, controlled by a temperature

parameter. As an analogy to the cooling process, the temperature parameter decreases as the execution progresses. Simulated annealing was successfully used to tackle the CVRP, e.g., [136, 116].

Iterated Local Search. It consists of alternating between a perturbation function to escape from local minima and a local search to explore the neighborhood of the perturbed solution. Many methods that achieve state-of-the-art results are based on this metaheuristic. A strategy similar to the petal algorithm was able to tackle many variants of the VRP. It is based on an iterated local search which generates a set of routes, and a set partitioning formulation which chooses the best combination of routes [170]. Other recent works include a hybridization between iterated local search and path relinking algorithm [125], and FILO, which uses iterated local search and a simulated annealing-based neighbor acceptance criterion to ensure diversification [1].

Greedy Randomized Adaptive Search Procedure (GRASP). This metaheuristic proposes to use a greedy construction heuristic with some degree of randomness to diversify the solution obtained, and an improvement moves to get a better solution. This metaheuristic has been used along with a path-relinking algorithm to tackle the CVRP [169]. A GRASP with multiple greedy construction heuristics and a Circle Restricted Local Search moves successfully tackled instances up to 200 clients [123].

Population-based algorithms. These methods maintain and improve over a set of candidate solutions. Most of these algorithms use nature-inspired concepts such as evolutionary algorithms or swarm intelligence.

Genetic Algorithm. This metaheuristic belongs to the evolutionary algorithms. Its first successful application to the VRP is due to Prins [149]. In their work, a solution is represented as a giant tour, without the depot as a trip delimiter. A shortest-path procedure, called Split, is used to delimit the routes by inserting the depot. Genetic operators are then used into the giant tour, and a local search is performed on the solution obtained after Split. The giant tour representation was later on used in Hybrid Genetic Search (HGS) of Vidal et al. [182, 183] where an efficient diversity control management is performed to not obtain a premature convergence of the population. This is done by introducing a fitness function

as a weighted sum of the solution cost and the Hamming distance to the other solutions. Vidal later on released an open-source implementation of HGS for the CVRP (HGS-CVRP) which is competitive with the state-of-the-art metaheuristic [181].

Ant Colony Optimization. This type of swarm intelligence uses agents called ants that construct solutions to the problem and records information about their quality in a joint memory. The memory is updated, so that paths leading to good quality solutions are favored. This mechanism is similar to the way ants put pheromones on the promising paths towards food. The D-Ants algorithm uses the Savings along with the pheromone information to build solutions for the CVRP, followed by a local search improvement. The algorithm showed remarkable results on instances up to 200 clients [152]. An Improved ant colony optimization algorithm that uses a mutation and an ant-weight strategy based on the candidate solution's overall cost and route cost for pheromone updates, revealed to be competitive with Tabu Search and Simulated Annealing for the CVRP [200]. For additional applications of Ant colony optimization for the routing problems, readers may refer to [44].

1.2 Machine learning concepts

In this section, we will introduce the key ideas in machine learning that we will use throughout this manuscript. We will cover different modern neural network architectures, as well as reinforcement learning.

1.2.1 Deep neural networks

1.2.1.1 Artificial neural networks

Artificial neural networks take their inspiration from the biological neural networks in the nervous system of animals. They were created to simulate the learning process of a biological brain. They are made of simple computational units called neurons.

Frank Rosenblatt's perceptron is the first artificial neural network, composed

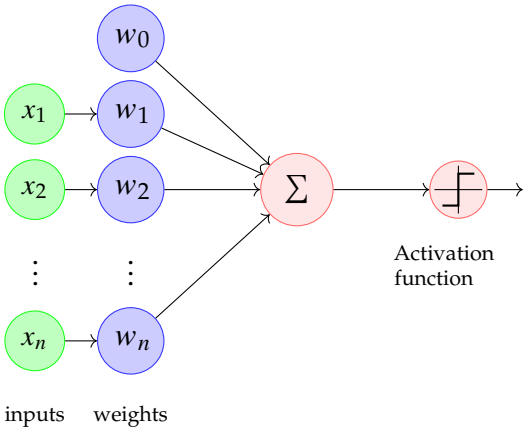


Figure 1.3: The perceptron model.

of a single formal neuron, created in 1958 [155]. This artificial neuron performs a weighted sum between input features followed by an activation function $\sigma(\cdot)$ (see Figure 1.3). Let $\mathbf{x} = [x_1, x_2, \ldots, x_n]^\intercal$ be a column vector of n features, and $\mathbf{w} = [w_1, \ldots, w_n]^\intercal$ a column vector of n weights, a perceptron implements the following function:

$$\begin{aligned} f(x) &= \sigma\Big(\sum_{i=1}^{n} w_i \cdot x_i + w_0\Big) \\ &= \sigma\Big(\mathbf{w}^\intercal \mathbf{x} + w_0\Big) \end{aligned}$$

where $\sigma(\cdot)$ is the Heaviside (sign) function, i.e.

$$H(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

The perceptron is mainly used as a linear classifier, where data can be separated using a hyperplane. Thus, the model is simple and is not suitable for processing complex data that cannot be separated linearly. The perceptron is then called a *shallow* network.

The multi-layer perceptron (MLP), is a type of feed-forward neural network, that generalizes the perceptron with more than one neuron per layer, and possibly

with more than one layer. Figure 1.4 depicts the general structure of a MLP. It is represented by a directed acyclic graph[6] with an input layer, hidden layers and an output layer with an activation function in between each layer. If we have several hidden layers, we speak of a *deep* neural network (DNN). The number of hidden layers is the **depth** of the neural network, and the number of neurons in a layer, is its **width**. Each neuron in the hidden and output layers performs an operation similar to the perceptron. Considering a layer i composed of m neurons and with an input $\mathbf{x_i} \in \mathbb{R}^d$, it performs the following operation: $f_i(\mathbf{x_i}) = \sigma\left(\mathbf{W_i}\mathbf{x_i}^\intercal + \mathbf{b_i}\right)$, where $\mathbf{W_i} \in \mathbb{R}^{m \times d}$, $\mathbf{b_i} \in \mathbb{R}^m$ are the weights and the bias of the layer, respectively, and $\sigma(\cdot)$ is the activation function[7]. The weights of a layer are grouped in a weight matrix, with m rows and n columns. Each row corresponds to a neuron in the layer, whereas each value in a column denotes the weight value pertaining to a specific input feature.

$$\mathbf{W_i} = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1d} \\ w_{21} & w_{22} & \dots & w_{2d} \\ \vdots & \ddots & \dots & \vdots \\ w_{m1} & w_{m2} & \dots & w_{md} \end{pmatrix}$$

The bias is organized in a column vector that is used to shift the result of each neuron's weighted sum, i.e. $\mathbf{b_i} = [b_1, \dots, b_m]^\intercal$.

A multi-layer perceptron, is then a stack of k layers that correspond to a composition of functions f_i. We denote this composition as

$$f(\mathbf{x}, \theta) = (f_1 \circ f_2 \circ \dots \circ f_k)(\mathbf{x}, \theta)$$

where θ regroups all the weights and biases ($\theta = \{\mathbf{W_1}, \mathbf{b_1}, \dots, \mathbf{W_k}, \mathbf{b_k}\}$) of the MLP.

The activation functions in the hidden layers add non-linearity to the MLP, which makes it able to approximate *any* real value function [78]. Thus, a MLP can separate complex data. Common activation functions include

- sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$,

[6] thus the name feed-forward neural network.

[7] Let us note that these activation functions are defined for any real value. Thus, for a vector $\mathbf{x} \in \mathbb{R}^d$, the activation function is applied element-wise as follows: $\sigma(\mathbf{x}) := [\sigma(x_1), \dots, \sigma(x_d)]$

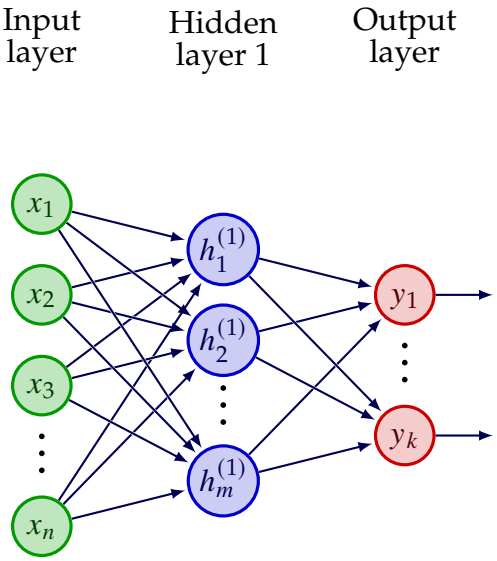
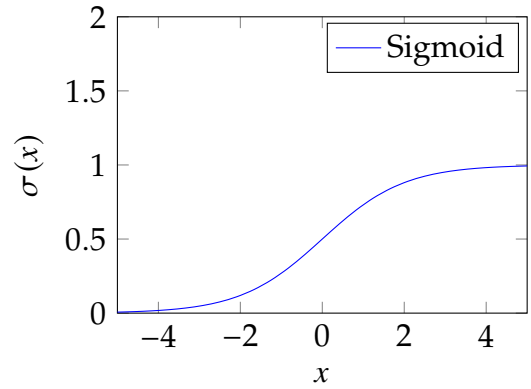


Figure 1.4: The multi-layer perceptron model.

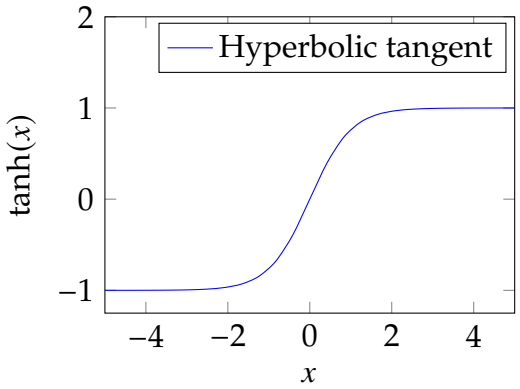
- hyperbolic tangent $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$,
- rectified linear units $ReLU(x) = \max(0, x)$,
- leaky rectified linear units $LReLU(x) = \begin{cases} 0.01 \cdot x & , x \leq 0 \\ x & , x > 0 \end{cases}$.

Figure 1.5 depicts the plots of the main activation functions in neural networks.

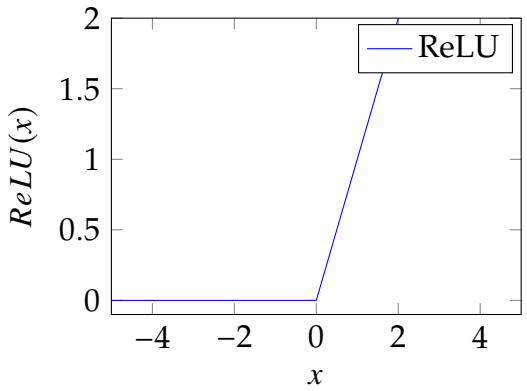
Traditional machine learning models have limited abilities in processing data in their raw form. They require domain experts to design feature extractors that transform raw data into feature vectors suitable for the task. Deep neural networks perform automatic feature extraction, i.e., they propose to discover the set of features, called representations, needed to perform the target task during the learning process from raw data. This is referred to as representation learning. The neural networks learn the representations hierarchically. The deeper the network, the more complex the learned representations [109]. Thus, a deep neural network is capable of transforming raw input data into an alternative representation in a feature space. This alternative representation is known as an **embedding**.



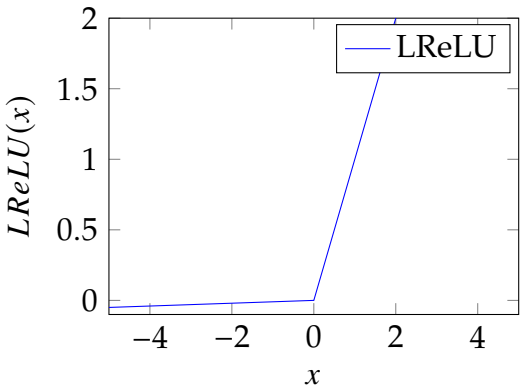
(a) Sigmoid activation function.



(b) Hyperbolic tangent activation function.



(c) ReLU activation function.



(d) Leaky ReLU activation function.

Figure 1.5: Common activation functions used in deep neural networks.

Training neural networks Neural network training involves an iterative process of forward pass, backward pass, and weight updates. In each training iteration, input data is fed forward through the network, generating predictions that are compared with the true values when dealing with supervised learning. The error is then computed using a loss function, and backpropagation is employed to calculate the gradients and propagate them backward through the network. The loss function depends on the types of predictions, which can either be discrete ones (classification) or continuous ones (regression).

In classification tasks, the goal is to predict discrete class labels for input data. Thus, each data point is assigned to a predefined category. The loss function commonly used in classification is the cross-entropy loss, which is defined for N samples and C classes as follows:

$$\mathcal{L}(\hat{\mathbf{Y}}, \mathbf{Y}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_{CE}(\hat{\mathbf{Y}}^{(i)}, \mathbf{Y}^{(i)})$$

where:

$$\mathcal{L}_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{j=1}^{C} \mathbf{y}_j \log(\hat{\mathbf{y}}_j)$$

$\hat{\mathbf{Y}}$ is a $N \times C$ matrix containing the predicted probability distribution over the classes for all N samples, where each row corresponds to the predicted probabilities for a single data point, and each column represents the probability of that data point belonging to a specific class. $\mathbf{Y}$ is also an $N \times C$ matrix, but it contains the one-hot encoded ground truth labels for each data point, indicating the true class for each sample. $\hat{\mathbf{y}}$ is a row vector representing the predicted probabilities for each class for a sample, $\mathbf{y}$ is a one-hot encoding row vector of the true label of the sample[8].

When classification tasks involve multiple classes, the softmax function is often used to produce class probabilities. The softmax function converts raw model outputs into probabilities:

$$\hat{\mathbf{y}}_i = softmax(e_i) = \frac{exp(e_i)}{\sum_{j=1}^{C} exp(e_j)}, \quad \forall i \in \{1, \ldots, C\}$$

[8]the vector has a value of 1 in the position corresponding to the class label and 0 in all other positions.

where $e_i \in \mathbb{R},\ i \in \{1, \ldots, C\}$ are the model's raw outputs. Let us note that the $softmax$ function takes a vector of C input values. To simplify notation, we abuse notation and represent the $softmax$ function applied to the i^{th} element of a vector $\mathbf{e} = [e_1, \ldots, e_C]$ as $softmax(e_i)$.

Regression tasks aim to predict continuous numerical values. In regression, the mean squared error (MSE) is a frequently used loss function:

$$\mathcal{L}_{MSE}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{N} \sum_{i=1}^{N} (\hat{\mathbf{y}}_i - \mathbf{y}_i)^2$$

where $\hat{\mathbf{y}}$ denotes the predicted values, $\mathbf{y}$ represents the true target values, and N is the total number of samples.

Neural networks use the gradient descent algorithm to update their weights and biases values[9]. It is an optimization algorithm that iteratively updates the weights of a neural network to minimize the loss $\mathcal{L}$. The weight update equation for gradient descent can be expressed as (with η being the learning rate):

$$\theta_i \leftarrow \theta_i - \eta \frac{\partial \mathcal{L}}{\partial \theta_i}$$

The standard gradient descent updates the neural network's parameters after evaluating the gradient of the loss function with respect to each individual training example. This approach is time-consuming and may lead to slow convergence. Therefore, to address these issues, the neural network parameters' update is performed by considering an accumulation of the gradient of the loss regarding a *mini-batch* of training examples. The gradients are averaged over the mini-batch before performing an update. A manually set *batch size* determines how many training examples are considered for each update.
Today's gradient descent optimizers are more sophisticated. For example, Adam which is one of the most effective and widely used optimization algorithms, uses an adaptive learning rate and maintains a memory of past gradients to accelerate the convergence and improve the training process [96].

To compute the partial derivative of the loss regarding the neural network's

[9] In this case, it must be fully differentiable. For example, the Heaviside function cannot be used as an activation function.

weights, the backpropagation algorithm is used. The algorithm uses the chain rule to compute the gradients of the weights in a backward pass from the output to the input layer.

To train the neural network, the dataset is typically divided into three subsets: the training set, the validation set, and the test set. The training set is used to train the neural network by exposing it to input data and their corresponding target values. The network updates its parameters based on the error calculated using the training set. The validation set serves as an independent dataset to assess the network's performance during training. It is used to monitor the model's generalization, and avoid overfitting: a phenomenon in machine learning where a model performs well on the training data but fails to generalize accurately to unseen data. For this, we measure the accuracy of the model on the validation set, i.e., the proportion of correct predictions made by the model over the total number of predictions, as measured on the validation set. The test set, which is separate from the training and validation sets, is used to evaluate the final performance of the trained model on unseen data. It provides an unbiased estimation of the model's predictive ability.

Finally, before diving into more complex neural network architectures, we give an example of how a MLP can be used to tackle a TSP instance. Figure 1.6 depicts an example of the first 3 steps of using a MLP to tackle a TSP instance with 5 cities $[\mathbf{x}_1, \ldots, \mathbf{x}_5]$. Each city is represented by its 2D-coordinates $\mathbf{x}_i = (x_i, y_i),\ i \in \{1, \ldots, 5\}$. Let's consider that the input of the network is the concatenation of all cities coordinates $[\mathbf{x}_1; \ldots; \mathbf{x}_5]$[10] and that we want to generate a categorical probability distribution such as the output is defined as $\hat{\mathbf{y}} \in [0, 1]^5$ and $\sum_{i=1}^{5} \hat{\mathbf{y}}_i = 1$. Each component $\hat{\mathbf{y}}_i$ will define the probability that the city i is the next best one to visit. At each step, a score $e_i \in \mathbb{R},\ i \in \{1, \ldots, 5\}$ is computed using the MLP. The scores are converted into a probability distribution using the $softmax$ function. If we suppose that we select greedily the city with the highest probability, at step 1, $\mathbf{x}_2$ is the first city that will be selected. At step 2, since $\mathbf{x}_2$ is already in the tour, we force its score to minus infinity so that when we use the $softmax$, its corresponding probability would be zero. Since $\mathbf{x}_1$ has the highest probability, it is selected as the next city to be visited. At step 3, $\mathbf{x}_1$ and $\mathbf{x}_2$ have their score set

[10] We denote the concatenation operation with $[\cdot;\cdot]$.

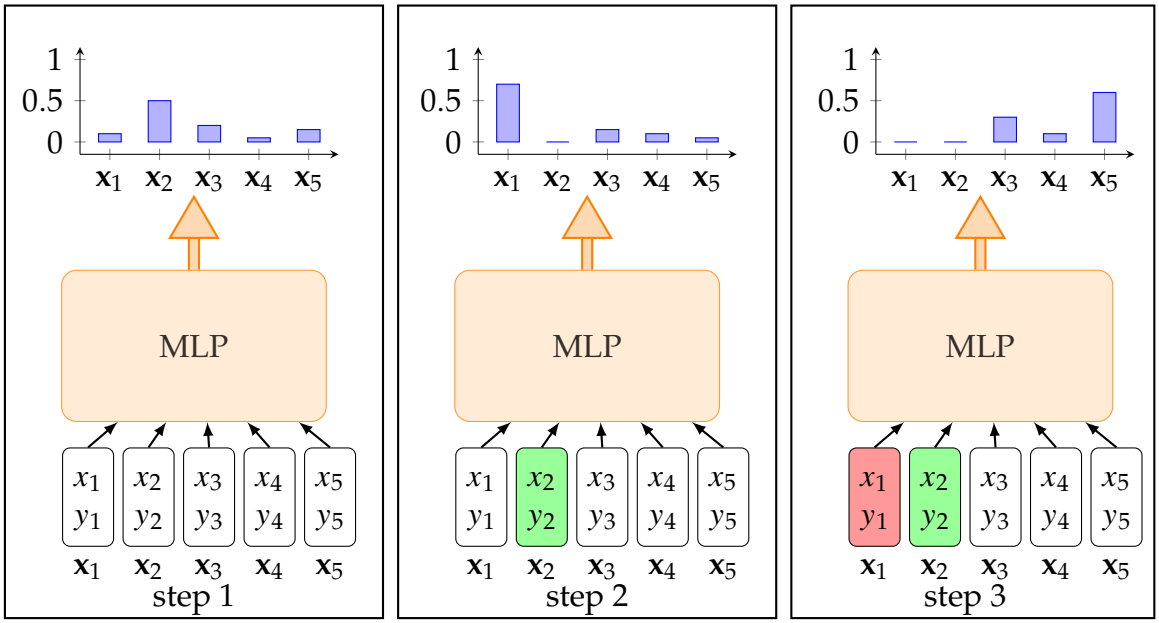


Figure 1.6: Example of the first 3 steps of using a MLP to solve a TSP instance.

to minus infinity. The city that is selected to extend the tour is $\mathbf{x}_5$. The process continues until all cities are visited.

Although a multi-layer perceptron can theoretically approximate any function given a sufficient number of hidden units and layers, in practice, depending on the type of data needed to be processed, there may be more suited neural network architectures. In the upcoming subsections, we will introduce the main ones that will be used in our work for tackling the VRP.

1.2.1.2 Recurrent neural networks

When dealing with raw inputs that are organized as a single vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^\intercal \in \mathbb{R}^n$, and we want to predict a single value of interest, a multi-layer perceptron can be used. However, when we have a sequence of τ raw inputs, like the coordinates of cities in the TSP, i.e. $\mathbf{x} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_\tau]$ with $\mathbf{x}_i = (x_i, y_i)^\intercal \in \mathbb{R}^2$, using a multi-layer perceptron is not sufficient to learn a good representation of the instance. A MLP would simply compute the representations of the cities independently of one another or mix all coordinates, as in the example of Figure

1.6. Yet, for a city, both considering information on other cities representations and considering specific information about the city in an isolated manner would be useful to have a better representation. Thus, it appears that another type of neural network would be more appropriate for this type of task. When input data is organized as a sequence, recurrent neural networks (RNNs) was the *de facto* option of choice at the beginning of our work, in 2019.

RNNs use a hidden state $\mathbf{h}_{t-1}$ to store the sequence information until the step $t-1$. The basic computation unit of RNNs is called a recurrent cell (see Figure 1.7). To compute the hidden state at step t, the previous hidden state $\mathbf{h}_{t-1} \in \mathbb{R}^{d_h}$ is used along with the current input $\mathbf{x}_t \in \mathbb{R}^n$, i.e. $\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1})$. An output y_t can then be computed using the hidden state $\mathbf{h}_t$ with a function $g(\mathbf{h}_t)$. The output y_t can then be used for prediction. The functions $f(\cdot)$ and $g(\cdot)$ are parametric functions that contain the weights and biases that will be learned. The simplest implementation considers two multi-layer perceptrons for f and g.

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}) = \sigma(\mathbf{W}_{hh}\ \mathbf{h}_{t-1} + \mathbf{W}_{hx}\ \mathbf{x}_t + \mathbf{b}_h)$$

$$\hat{y}_t = g(\mathbf{h}_t) = \sigma(\mathbf{W}_{hy}\ \mathbf{h}_t + \mathbf{b}_y)$$

where $\mathbf{W}_{hh} \in \mathbb{R}^{d \times d_h}$, $\mathbf{W}_{hx} \in \mathbb{R}^{d \times n}$, $\mathbf{W}_{hy} \in \mathbb{R}^{d_y \times d}$ are the weights matrices, and $\mathbf{b}_h \in \mathbb{R}^d$, $\mathbf{b}_y \in \mathbb{R}^{d_y}$ are the bias vectors.

Simple RNN cells are generally hard to train. They are subject to the vanishing gradient phenomenon, which causes the weights of the RNN to perform little or no updates during backpropagation because of the partial derivatives that are close to zero. To prevent this, more sophisticated recurrent cells have been designed, namely Long Short-Term Memory (LSTM) [76] and Gated Recurrent Unit (GRU) [31].

Recurrent neural networks can be used on various tasks, that we can group into three classes.

1. **Many-to-one.** This type corresponds to using an input of sequential features, and outputting a single value (see Figure 1.8). The penultimate hidden state $\mathbf{h}_{\tau-1}$ encodes information about all previous inputs $\{x_i\}_{i=1}^{\tau-1}$. Along with the last input x_τ, it outputs a single value y which corresponds to the target task

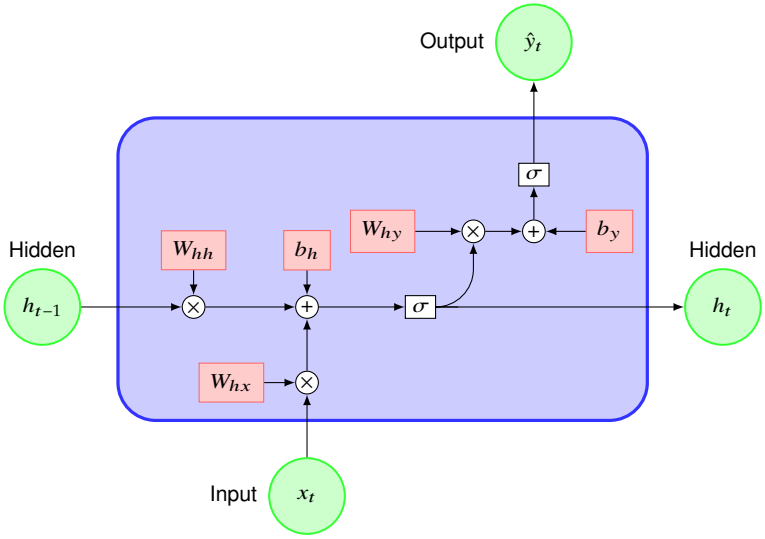


Figure 1.7: A simple Recurrent Neural Network cell.

prediction. For example, we can input the set of cities' coordinates of a TSP instance and output a scalar that estimates the optimal tour length.

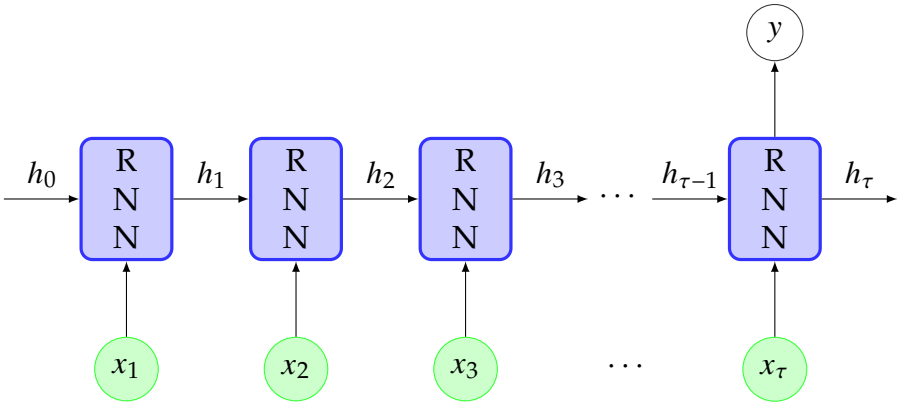


Figure 1.8: Input-to-output mapping: many-to-one.

2. **Many-to-many.** Tasks of this type are encountered when mapping a sequence into another sequence (see Figure 1.9). An input $\{x_i\}_{i=1}^{\tau}$ and hidden state $\{\mathbf{h}_{i-1}\}_{i=1}^{\tau}$ are used to predict an element of the target task $\{y_j\}_{j=1}^{m}$. In the figure, the input and output are of the same length, but in general $m \geq \tau$. An example of a task of this type is learning to output a solution to the

traveling salesman problem. The cities' coordinates are given one at a time to the RNN, which outputs the position of the city in the permutation.

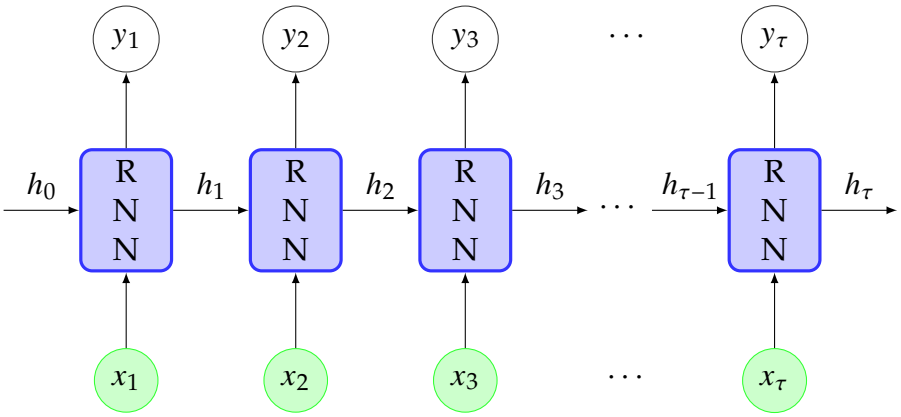


Figure 1.9: Input-to-output mapping: many-to-many.

3. **One-to-many.** This involves tasks that use a single input to generate a sequence of outputs (see Figure 1.10). For instance, this can be used to generate an image caption. The input is a single vector of image features, while the output is a sequence of words representing the caption.

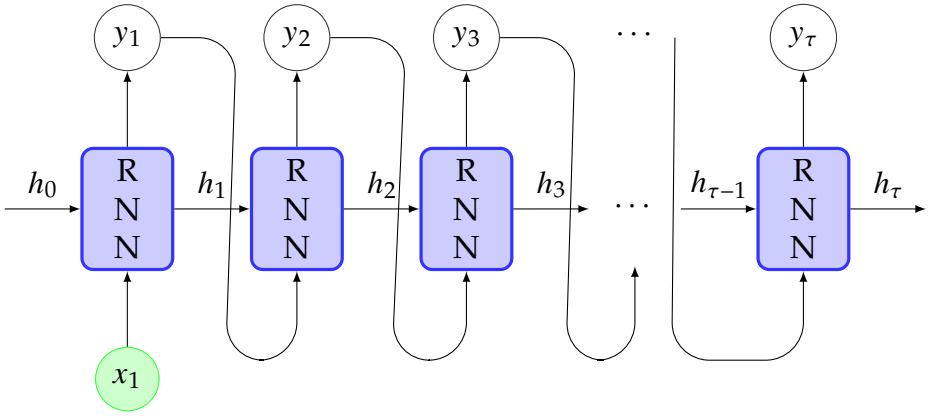


Figure 1.10: Input-to-output mapping: one-to-many.

Encoder-Decoder architecture. When tackling a task that requires to map a sequence into another sequence, chances are that it will be tackled as a many-to-many one. However, by construction, this configuration only takes the previous

hidden state and the current element of the sequence as input. It does not take into consideration the previous output, while this may be valuable information to fully grasp the context of resolution. Another choice may be to use a one-to-many configuration which makes predictions according to the last predicted value. The drawback is that, it needs a single input value. To overcome these issues and take the best of both worlds, an Encoder-Decoder model has been proposed [32]. Two neural networks are used in this setting. The first neural network, called Encoder, is responsible for mapping the input sequence into a single vector representation. The second neural network, called the Decoder, produces a sequence using the encoder's vector representation as the initial input, then it uses the first decoder's output as input. This new architecture improves the results obtained on several learning problems. It has become the state-of-the-art architecture used to tackle several problems, including VRPs (e.g., [185]). Figure 1.11 shows an example of an Encoder-Decoder architecture made of two RNNs. However, let us note that the Encoder and the Decoder neural networks can be of different types.

1.2.1.3 The attention mechanism

The attention mechanism is probably one of the most important concepts in deep learning that has resulted in breakthrough in many difficult tasks. It is inspired from the biological attention of animals. Indeed, when they receive a massive amount of data from their environment, animals can focus on a part of the information. For example, when humans read books, they do not process the entire page at once, but they concentrate on the specific sentence that they are reading. The attention mechanism in deep neural networks mimics this behavior by retaining only part of the input that is being processed.

The attention mechanism computes attention scores $\{e_i\}_{i=1}^{n}$ between keys $\mathbf{K} \in \mathbb{R}^{n \times d_k}$ and a query $\mathbf{q} \in \mathbb{R}^{d_q}$ using an energy function E, i.e. $e_i = E(\mathbf{q}, \mathbf{K}_i)$, $\forall i \in \{1, \ldots, n\}$, with n being the set length [133]. The keys are the representations, computed by an encoder, of each input of a set of length n each represented in a d_k dimension space, i.e. $\mathbf{K} = [\mathbf{k_1}, \mathbf{k_2}, \ldots, \mathbf{k_n}]$, with $\mathbf{k_i} \in \mathbb{R}^{d_k}$ being row vectors of the matrix. The query $\mathbf{q}$ is generally task-dependent. The vector can be seen as literally a query or a question formulated by a model to ask for the relevant parts

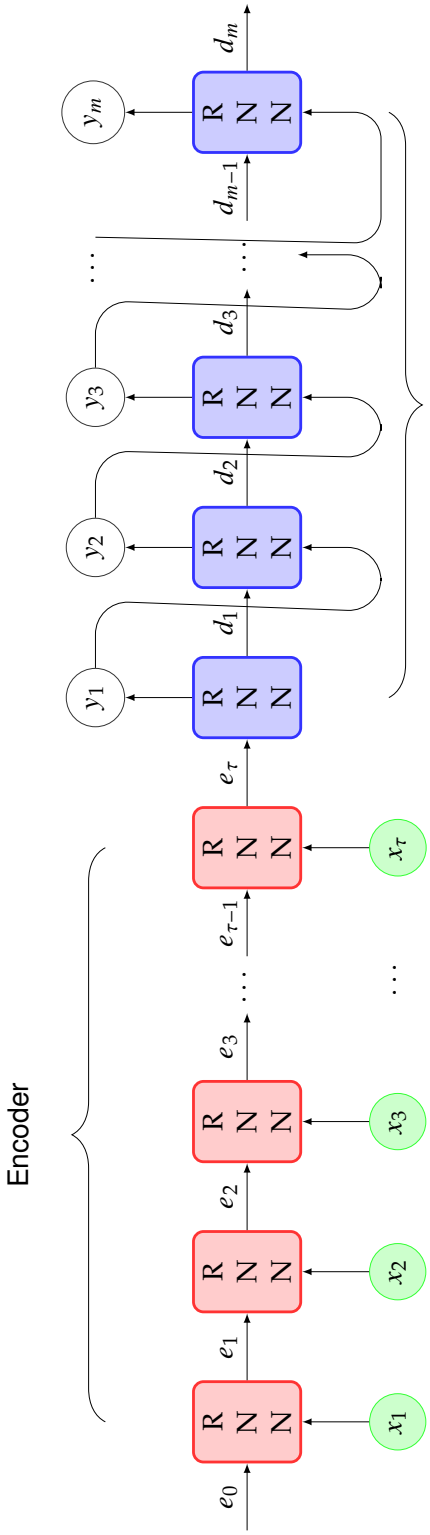


Figure 1.11: Encoder-Decoder architecture for tackling a many-to-many task.

of the input necessary to perform the task.

The energy function reflects the correlation between the query and the keys. The higher the correlation, the more relevant will be the corresponding key. Various energy functions have been designed in the literature [133]. We summarize them in Table 1.2 that follows. The most commonly used attention mechanisms are the additive and multiplicative ones. An empirical comparison between these two mechanisms showed that additive attention slightly outperforms the multiplicative one [25]. However, the multiplicative attention is computationally less expensive to compute. The scaled dot-product is a variant of the multiplicative attention, where a scaling factor $\frac{1}{\sqrt{d_k}}$ is introduced to overcome the vanishing gradient problem of the softmax function [178]. The general attention scoring function is a generalization of the multiplicative attention, where a linear transformation of the keys is introduced. Finally, the cosine attention uses a cosine similarity to compute a score between the keys and the query. Let us note that the multiplicative, scaled dot-product and the cosine similarity attentions require that $d_k = d_q$.

Energy function	Formula
Additive [25]	$E(\mathbf{q}, \mathbf{k}) = \mathbf{v}^\intercal \cdot tanh(\mathbf{W_q} \cdot \mathbf{q} + \mathbf{W_k} \cdot \mathbf{k})$
Multiplicative (dot-product) [25]	$E(\mathbf{q}, \mathbf{k}) = \mathbf{q} \cdot \mathbf{k}^\intercal$
Scaled dot-product [178]	$E(\mathbf{q}, \mathbf{k}) = \frac{\mathbf{q} \cdot \mathbf{k}^\intercal}{\sqrt{d_k}}$
General [119]	$E(\mathbf{q}, \mathbf{k}) = \mathbf{q} \cdot \mathbf{W} \cdot \mathbf{k}^\intercal$
Cosine similarity [68]	$E(\mathbf{q}, \mathbf{k}) = \frac{\mathbf{q} \cdot \mathbf{k}^\intercal}{\|\mathbf{q}\| \cdot \|\mathbf{k}\|}$

Table 1.2: Different energy functions used to compute the attention scores. $\mathbf{q} \in \mathbb{R}^{d_q}, \mathbf{k} \in \mathbb{R}^{d_k}, \mathbf{v} \in \mathbb{R}^{d}; \mathbf{W_q} \in \mathbb{R}^{d \times d_q}, \mathbf{W_k} \in \mathbb{R}^{d \times d_k}, \mathbf{W} \in \mathbb{R}^{d_q \times d_k}$ are learnable parameters of the energy function. $\| \cdot \|$ is the Euclidean norm.

The scores $\{e_i\}_{i=1}^n$ are then converted into attention weights through an attention distribution function. Generally, the softmax function is used for that

purpose.

$$\alpha_i = softmax(e_i) = \frac{exp(e_i)}{\sum_{j=1}^{n} exp(e_j)}, \quad \forall i \in \{1, \ldots, n\}$$

Since $\sum_{i=1}^{n} \alpha_i = 1$ and $0 \leq \alpha_i \leq 1, \ \forall i \in \{1, \ldots, n\}$, thus, the attention distribution can be viewed as a categorical distribution from which we can sample an element accordingly. As we will see, this property is later used in designing deep neural network methods for solving combinatorial optimization problems [185]. Moreover, the attention mechanism has allowed new developments for neural network architectures, as we will see in the next subsection.

1.2.1.4 Transformer

One of the limitations of RNNs is that the sequential processing of the input data cannot be parallelized, since to compute the current hidden state $\mathbf{h}_t$, it is needed to have the previous hidden state $\mathbf{h}_{t-1}$. Furthermore, empirical evaluations show that it is difficult to process very long sequences with recurrent neural networks due to the vanishing gradient phenomenon. In addition, RNNs consider sequences, which implies that the order in which the data is passed to the neural network is important. That may not be suited for some tasks, such as the TSP. This has motivated research for alternatives that can overcome these shortcomings. In 2017, a team at Google developed Transformers, a new type of feed-forward network based primarily on self-attention [178].

A Transformer is a stack of N identical transformer blocks, where each output of a block in the input of the next block. A Transformer block is made of two sublayers: a multi-head attention layer ($MHA(\cdot)$) and a feed-forward layer ($FF(\cdot)$) (see Figure 1.12). Between each sublayer, there is a skip connection and a layer normalization to prevent overfitting and to speed up training [10], i.e. for a set of input vectors $\{\mathbf{x}_i\}_{i=1}^{n}$ (with $\mathbf{x}_i \in \mathbb{R}^d$):

$$\mathbf{o}_i = LayerNorm(\mathbf{x}_i + MHA(\mathbf{x}_i)), \quad \forall i \in \{1, \ldots, n\}$$

$$\mathbf{h}_i = LayerNorm(\mathbf{o}_i + FF(\mathbf{o}_i)), \quad \forall i \in \{1, \ldots, n\}$$

Skip connection [72] allows the output to bypass one or more layers and to be

added to the final output, i.e, $\mathbf{x}_{i+1} = f(\mathbf{x}_i) + \mathbf{x}_i$. This enables the design of deep architectures without degrading the performance of the deep neural network.
The layer normalization [10] is a type of normalization that operates on a single training example and normalizes the values across each feature independently. For each training example, the mean, and variance are calculated for each feature, and then the features are normalized based on these statistics. This normalization improves the convergence and the generalization of the neural network.

The Multi-Head Attention $MHA(\cdot)$. As introduced in the previous subsection, the attention scores are computed using a query and keys. Transformer extends this idea by considering the attention as a mapping from queries to key-value pairs. The idea behind this, is to be able to compute new representations for each input as a convex sum of all the values, with different coefficients computed using the self-attention mechanism. Here, *"self"* refers to the fact that this mechanism is used on data that comes from the same set ; if we consider a single input as a set of vectors $\{\mathbf{x}_i\}_{i=1}^n$, then each vector $\mathbf{x}_i$ attends to another input vector $\mathbf{x_j}$ ($j \in \{1, \dots, n\}$). To diversify the representations, the attention mechanism is decoupled L times, hence multiple queries and key-value pairs are computed for the same inputs; therefore it uses multiple attention heads (thus the name Multi-Head Attention). A scaled dot-product is used to compute the attention scores, followed by a softmax activation function, i.e.

$$\alpha_{ij}^l = softmax\Big(\frac{\mathbf{q}_i^l \cdot (\mathbf{k}_j^l)^\intercal}{\sqrt{d_k}}\Big), \quad \forall i, j \in \{1, \dots, n\}, \forall l \in \{1, \dots, L\} \tag{1.1}$$

The new head-wise representation $\mathbf{o}_i^l$ is computed using a convex sum of all the values $\{\mathbf{v}_j^l\}_{j=1}^n$ using the attention scores.

$$\mathbf{o}_i^l = \sum_{j=1}^{n} \alpha_{ij}^l \ \mathbf{v}_j^l, \quad \forall i \in \{1, \dots, n\} \tag{1.2}$$

where $\forall i \in \{1, \dots, n\}, \ \forall l \in \{1, \dots, L\}$:

$$\mathbf{q}_i^l = \mathbf{W}_q^l \ \mathbf{x}_i + \mathbf{b}_q^l$$

$$\mathbf{k}_i^l = \mathbf{W}_k^l \ \mathbf{x}_i + \mathbf{b}_k^l$$

$$\mathbf{v}_i^l = \mathbf{W}_v^l \ \mathbf{x}_i + \mathbf{b}_v^l$$

with $\mathbf{x}_i \in \mathbb{R}^{d_x}$, $\mathbf{q}_i^l \in \mathbb{R}^{d_q}$, $\mathbf{k}_i^l \in \mathbb{R}^{d_k}$, $\mathbf{v}_i^l \in \mathbb{R}^{d_v}$, $\mathbf{W}_q^l \in \mathbb{R}^{d_q \times d_x}$, $\mathbf{W}_k^l \in \mathbb{R}^{d_k \times d_x}$, $\mathbf{W}_v^l \in \mathbb{R}^{d_v \times d_x}$, $\mathbf{b}_q^l \in \mathbb{R}^{d_q}$, $\mathbf{b}_k^l \in \mathbb{R}^{d_k}$, $\mathbf{b}_v^l \in \mathbb{R}^{d_v}$ and $q_k = d_q$.

The head-wise representations $\mathbf{o}_i^l$ are then concatenated together to form the final layer representation $\mathbf{o}_i$.

$$\mathbf{o}_i = [\mathbf{o}_i^1; \mathbf{o}_i^2; \ldots; \mathbf{o}_i^L], \quad \forall i \in \{1, \ldots, n\}$$

Since there is a skip connection, right after the MHA function, we must keep the dimension of o_i the same as the dimension of $\mathbf{x}_i$, i.e. $\mathbf{o}_i \in \mathbb{R}^{d_x}$. To do so, we choose the number of heads as a divisor of d_x ($d_x \mod L = 0$).

The feed-forward layer $FF(\cdot)$. It is simply a multi-layer perceptron with two hidden layers and an activation function in between, i.e.

$$FF(\mathbf{x}) = \mathbf{W}_2 \ ReLU(\mathbf{W}_1 \ \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

where $\mathbf{x} \in \mathbb{R}^{d_x}, \mathbf{W}_1 \in \mathbb{R}^{d_1 \times d_x}, \mathbf{b}_1 \in \mathbb{R}^{d_1}, \mathbf{W}_2 \in \mathbb{R}^{d_2 \times d_1}, \mathbf{b}_2 \in \mathbb{R}^{d_2}$.

One of the strong properties of the Transformer neural network is its ability to compute a representation for an input that is a set; thus it is invariant to the order of the elements in the input, contrary to RNNs that assume data to be a sequence and for which element ordering will influence the model's outputs. Thus, using a Transformer for VRPs does not introduce a bias regarding inputs' order.

Let us note that it is still possible to encode a sequence using Transformers by introducing a positional encoding [178]. Positional encoding is added to the input embeddings of the transformer model. It allows the model to differentiate between inputs based on their position, enabling it to consider the order of the sequence.

1.2.1.5 Graph neural networks

In the section 1.1, we saw that the VRP has a graph formulation. It is possible to take advantage of that formulation by considering a type of neural networks that

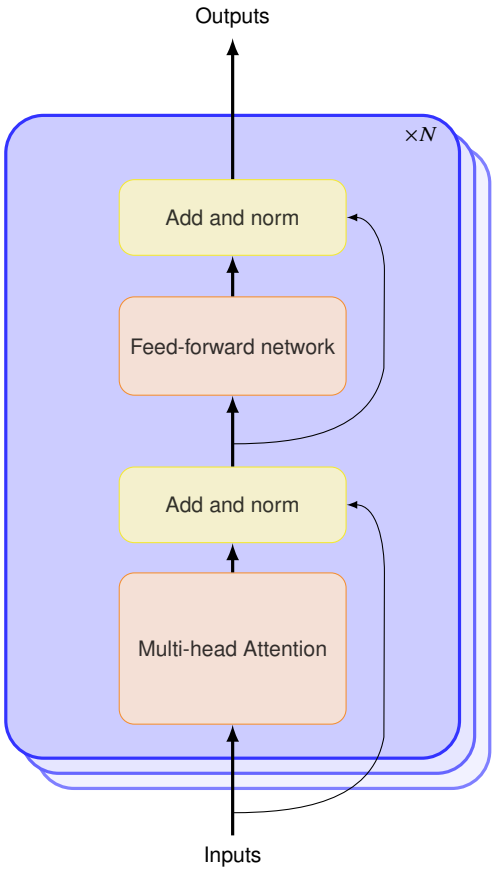


Figure 1.12: The Transformer model.

are adapted to graphs, namely Graph Neural Networks (GNNs).

The goal of GNNs is to compute the nodes and the graph representations based on the structure of the graph, i.e., starting from a set of node and edge features, GNNs define nodes embeddings and a graph embedding, that are a representation of, respectively, the nodes and the graph in a high dimension vector space. The *message-passing framework* that stacks K GNN layers is used for that [70]. Each layer $k \in \{1, \dots, K\}$ can be viewed as a differentiable function with parameters $\theta^{(k)}$ that computes for each node u a node embedding $\mathbf{h}_u^{(k)}$ as follows:

$$\mathbf{h}_u^{(k)} = F(\mathbf{h}_u^{(k-1)}, \{\mathbf{h}_v^{(k-1)}\}_{v \in \mathcal{N}_u}, \{a_{uv}\}_{v \in \mathcal{N}_u}, \theta^{(k)})$$

$\mathcal{N}_u$ is the set of the neighboring nodes of a node $u \in V$ and $\{a_{uv}\}_{v \in \mathcal{N}_u}$ the set of edge features of the edge set that link u to its neighbors $v \in \mathcal{N}_u$.

The function F itself relies on two mechanisms: neighborhood message aggregation and node embedding update (see Figure 1.13 for an example of an implementation of a message passing scheme), defined as:

$$\mathbf{m}_u^{(k)} = \textsc{Aggregate}\big(\{\mathbf{h}_v^{(k-1)}\}_{v \in \mathcal{N}_u}, \{a_{uv}\}_{v \in \mathcal{N}_u}\big)$$
$$\mathbf{h}_u^{(k)} = \textsc{Update}(\mathbf{h}_u^{(k-1)}, \mathbf{m}_u^{(k)})$$

The Aggregate function gathers the node and edge representations from the node's neighbors and their connecting edges, and computes an embedding that summarizes the gathered information. An example of aggregation is the sum of node embeddings, i.e. $\mathbf{m}_u^{(k)} = \sum_{v \in \mathcal{N}_u} \mathbf{h}_v^{(k-1)}$. Other aggregation functions include the max, mean, and weighted sum of the neighbors embeddings. The Update functions combines the embedding of the node u from layer $k-1$ with the message from the neighbors into a new node embedding $\mathbf{h}_u^{(k)}$. A simple Update function can consider passing the node embedding from the previous layer $\mathbf{h}_u^{(k-1)}$ and the neighbors message $\mathbf{m}_u^{(k)}$ through two different MLPs, summing up the result, and adding non-linearity with an activation function, i.e.

$$\textsc{Update}(\mathbf{h}_u^{(k-1)}, \mathbf{m}_u^{(k)}) = \sigma(MLP_1(\mathbf{h}_u^{(k-1)}) + MLP_2(\mathbf{m}_u^{(k)}))$$

We can distinguish two families of GNNs: spectral and spatial GNNs.

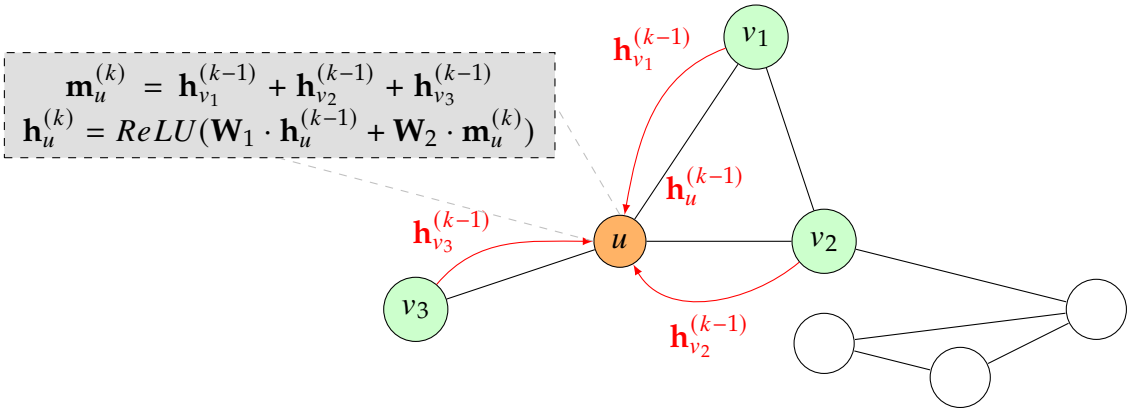


Figure 1.13: Example of a GNN layer that computes the embedding of the node u (in orange) using a sum for the aggregation of the embeddings of the neighboring nodes (in green) $\mathbf{h}_{v_1}^{(k-1)}, \mathbf{h}_{v_2}^{(k-1)}, \mathbf{h}_{v_3}^{(k-1)}$ and an update function made of two MLPs with weights $\mathbf{W}_1$ and $\mathbf{W}_2$ and `ReLU` activation function.

Spectral GNNs rely on spectral graph representations based on graph signal processing theory. Properties such as adjacency matrix, graph Laplacian, eigenvector and eigenvalue decomposition are exploited in order to compute the embeddings. Graph Convolutional Networks (GCN) is one of the most used neural networks of this type [97]. GCN operates the AGGREGATE and UPDATE functions in one by introducing self loops, and it uses a symmetric normalization based on the degree of each node, i.e.,

$$\mathbf{h}_u^{(k)} = \sigma\Big(\mathbf{W}^{(k)} \cdot \sum_{v \in \mathcal{N}_u \cup \{u\}} \frac{\mathbf{h}_v^{(k-1)}}{\sqrt{|\mathcal{N}_u| \cdot |\mathcal{N}_v|}}\Big)$$

Spatial GNNs exploits the graph topology by operating on a group of closed neighbors. One of the successful approaches of this type is the Graph Attention network (GAT) [179]. GAT uses a weighted sum of embeddings of a node, and its neighbors to compute a new node's embedding. It uses the Multi-head attention mechanism from Transformer to compute multiple embeddings of the same node, i.e., for each head l, the following computation is performed:

$$\mathbf{h}_u^{(k),l} = \sum_{v \in \mathcal{N}_u} \alpha_{uv}^l \cdot \mathbf{W}^{(k),l} \mathbf{h}_v^{(k-1),l}$$

where α_{uv}^{l} are computed using an attention mechanism:

$$\alpha_{uv}^{l} = \frac{exp(LReLU(\mathbf{a}^{\intercal}[\mathbf{W}^{(k),l}\mathbf{h}_{u}^{(k-1),l} \; ; \; \mathbf{W}^{(k),l}\mathbf{h}_{v}^{(k-1),l}]))}{\sum_{v' \in \mathcal{N}_u} exp(LReLU(\mathbf{a}^{\intercal}[\mathbf{W}^{(k),l}\mathbf{h}_{u}^{(k-1),l} \; ; \; \mathbf{W}^{(k),l}\mathbf{h}_{v'}^{(k-1),l}]))}$$

The final node embedding is represented either by concatenating or averaging the multi-head embeddings $\mathbf{h}_{u}^{(k),l}$, followed by an activation function $\sigma(\cdot)$; i.e.

$$\mathbf{h}_{u}^{(k)} = \sigma\Big([\mathbf{h}_{u}^{(k),1}; \mathbf{h}_{u}^{(k),2}; \ldots; \mathbf{h}_{u}^{(k),L}]\Big)$$

$$\mathbf{h}_{u}^{(k)} = \sigma\Big(\frac{1}{L}\sum_{l=1}^{L}\mathbf{h}_{u}^{(k),l}\Big)$$

with $\mathbf{W}^{(k),l} \in \mathbb{R}^{d \times d}$ are the weight matrices of the l^{th} head of the k^{th} GNN block for the node embeddings, $\mathbf{h}_{v}^{(k-1),l}$ is the node embedding from the l^{th} head of the $(k-1)^{th}$ GNN block, and $\mathbf{a} \in \mathbb{R}^{d}$ is a vector of learned parameters.

The literature around GNNs is rich and continues to grow, reader may refer to a review regarding different aspects of GNNs [208], or a recent book [194][11].

1.2.2 Reinforcement learning

1.2.2.1 Definition

Reinforcement learning is a subfield of Machine Learning that enables an agent to learn from its interactions with its environment. The agent takes a sequence of actions in the context of a sequential decision-making problem, e.g., solving a TSP requires selecting one city after another. In reinforcement learning, no instruction is given to the agent about how it must behave (act) to achieve its end goal, instead, only a scalar, called reward, is returned to the agent to tell it how good its actions were[12]. Essentially, reinforcement learning is learning by trial-and-error; the agent performs actions on the environment, then it receives a reward, and it adjusts its actions accordingly.

[11] Freely available at: `https://graph-neural-networks.github.io/`

[12] We introduce Reinforcement Learning as a reward maximization problem, but it can also be formulated as a cost minimization problem so that it fits the case of CVRP, for example.

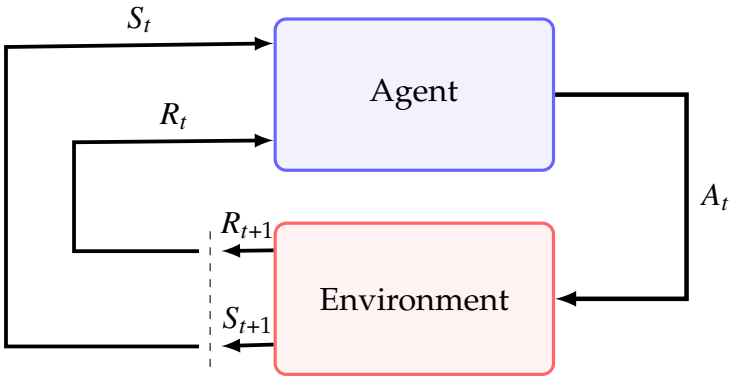


Figure 1.14: The agent-environment interaction diagram in Reinforcement Learning, adapted from Sutton and Barto [171].

Formally, we model reinforcement learning problems using a Markov Decision Process (MDP). An MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p, \gamma \rangle$ representing the set of states, actions, reward function, a transition probability function, and the discount rate, respectively. At each time step $t \in \{1, \ldots, T\}$[13], the agent observes a state $S_t \in \mathcal{S}$ and, on that basis, takes an action $A_t \in \mathcal{A}$. A reward R_{t+1} is returned by the environment to the agent, as well as the new state S_{t+1} (see Figure 1.14). Thus, reinforcement learning is an ideal framework to model sequential decision-making problems, where we have a *trajectory* given by the states, actions, and rewards as follows: $\mathcal{T} = S_0, A_0, R_0, S_1, A_1, R_2 \ldots$. When the trajectory reaches the terminal state S_T, in case of finite MDPs, we call it an *episode*.
The *dynamics* of the problem are defined by the state-transition probabilities, where the next state depends solely on the current state and action values (thus respecting the Markov property), i.e.,

$$\begin{aligned} p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} &\longrightarrow [0,1] \\ (s', s, a) &\longmapsto p(s'|s,a) = P(S_{t+1} = s' | S_t = s, A_t = a) \end{aligned}$$

The reward is a central quantity in reinforcement learning. Formally, rewards are computed using a reward function that depends on the current state, the action and the resulting next state, i.e. $R_t = \mathcal{R}(S_t, A_t, S_{t+1})$. The reward function can either be deterministic or stochastic, depending on the problem being tackled.

[13] In our case, we consider only *finite* MDP with time horizon $T \neq +\infty$.

The goal of the agent is to maximize the cumulative reward over a trajectory starting from time t until the time horizon T is reached, which is called *return*. The return G_t is generally discounted using a discount factor $\gamma \in [0, 1]$ i.e.,

$$G_t := \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$$

.

In reinforcement learning, the agent acts by following a *policy* denoted as $\pi(a|s)$. In general, the policy is stochastic, and is defined as a mapping from states to probabilities of selecting each possible action;

$$\pi(a|s) := P(A_t = a|S_t = s)$$

A good policy leads to high returns, while a bad one gives poor quality returns. If we note J_π as the expected return for the policy π, i.e., $J_\pi := \mathbb{E}_\pi[G_t]$, then a good policy is the one that maximizes the expected return among all possible policies, which corresponds to $\pi^* = \arg\max_\pi \mathbb{E}_\pi[G_t]$

There are two ways of solving a reinforcement learning problem:

1. Value-based methods: these methods estimate the value of states or state-action pairs. The agent uses the value function to derive a policy by selecting actions that maximize the value estimates.

2. Policy-based methods: instead of estimating the value of different actions or state-action pairs, policy-based methods focus on finding the best policy directly. The policy is typically represented as a parameterized function, such as a neural network.

1.2.2.2 Value-based methods

Value-based methods define *value* functions, which are functions that estimate "the goodness" of a given state or a state-action pair when following a given policy. This is determined by the expected return the agent can receive in a given state when acting according to the policy π. We can distinguish the *state value* function and the *state-action* value function.

$$V_\pi(s) := \mathbb{E}_\pi[G_t|S_t = s] \tag{1.3}$$

$$Q_\pi(s,a) := \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{1.4}$$

Thus, to solve reinforcement learning problems using value-based methods, we estimate the value functions. One central idea for that is to use Temporal Difference (TD) Learning. We estimate the state value and state-action value functions through iterative update as follows (α being the learning rate):

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{1.5}$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a \in \mathcal{A}(S_{t+1})} Q(S_{t+1}, a) - Q(S_t, A_t)) \tag{1.6}$$

The equation 1.6 gives us an ubiquitous value-based algorithm named Q-Learning [191]. The algorithm estimates the optimal state-action value function by iteratively updating the Q-values in an off-policy manner, i.e., the update is performed by choosing the action with the maximum return from S_{t+1}, which does not necessarily correspond to the policy that is actually learned (see Algorithm 1). To balance between exploration (trying new actions to explore the environment) and exploitation of actions that appear to be the best based on past knowledge, we can use an $\epsilon-$greedy strategy; with a probability ϵ, the agent selects a random action (exploration), and with a probability $1 - \epsilon$, the agent selects the action with the highest Q-value estimation (exploitation).

When the state and action spaces are huge, the computation of all the Q-values becomes intractable. To overcome this limitation, modern Q-learning algorithms use deep neural networks to estimate the Q-values (i.e., $Q(s, a; \theta)$), resulting in Deep Q-learning (DQN [130]). It uses an experience replay buffer $\mathcal{B}$ to store trajectories (s, a, r, s') and performs update by sampling from the buffer and minimizing a regression loss. The loss function in Deep Q-learning is typically defined using the Mean Squared Error (MSE) between the predicted Q-values and the

Algorithm 1: Q-learning algorithm for episodic MDP (adapted from Sutton and Barto [171])

Input: learning rate α, discount factor γ, exploration parameter ϵ
foreach *episode* **do**
  Initialize S_0
  foreach $t \in \{0, \ldots, T\}$ **do**
    Choose A_t from S_t using a policy derived from Q (e.g. $\epsilon-$greedy)
    Take action A_t, observe R_{t+1}, S_{t+1}
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a \in \mathcal{A}(S_{t+1})} Q(S_{t+1}, a) - Q(S_t, A_t))$
  end
end

target Q-values.

$$\mathcal{L}(\theta) := \mathbb{E}_{(s,a,r,s') \sim \mathcal{B}} \left[(y - Q(s, a; \theta))^2 \right]$$

The target Q-value y is computed as:

$$y = r + \gamma \max_{a'} Q(s', a'; \theta^-)$$

where $Q(s', a'; \theta^-)$ represents the maximum predicted Q-value over all possible actions a' in the next state s', using a separate target Q-network with parameters θ^-.

1.2.2.3 Policy-based methods

Policy-based methods propose to search for the optimal policy directly. The policy can either be a deterministic one or a stochastic one. Stochastic policies are favored when dealing with discrete action spaces.

Basically, stochastic policies work by increasing the probability value of actions that maximize the expected return. Generally, a parameterized function represents these policies π_θ. Thus, finding a good policy reverts to an optimization problem. The optimization is done by policy gradient algorithms on a policy $\pi_\theta(a|s)$ represented as a differentiable function with parameters θ. The search for a policy is therefore formulated as a continuous optimization problem where we aim to

maximize the objective function $J(\theta)$ defined as:

$$J(\theta) = \mathop{\mathbb{E}}_{\pi_\theta}[G_t]$$

The policy gradient theorem states that the gradient of the objective function with respect to the policy parameters θ can be expressed as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t\right] \tag{1.7}$$

The REINFORCE algorithm approximates the expected gradient in the policy gradient theorem using sampled trajectories [192]. The update equation for the policy parameters in REINFORCE is as follows:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) G_t,$$

where α is the learning rate.

Algorithm 2 presents the REINFORCE algorithm in the episodic case.

Algorithm 2: REINFORCE Algorithm (adapted from Barto and Sutton [171])

Input: Policy π_θ with parameters θ, learning rate α
Result: Learned policy π_θ
Initialize policy parameters θ randomly;
foreach *episode* **do**
  Generate an episode following π_θ:
  $\{(s_0, a_0, r_1), (s_1, a_1, r_2), \ldots, (s_{T-1}, a_{T-1}, r_T)\}$;
  foreach *time step* $t = 0$ *to* $T-1$ **do**
    Compute the return: $G_t \leftarrow \sum_{k=t}^{T} \gamma^{k-t} r_{k+1}$;
    Compute the policy gradient: $\nabla_\theta \log \pi_\theta(a_t|s_t)$;
    Update the policy parameters: $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) G_t$;
  end
end
return Learned policy π_θ;

REINFORCE algorithm relies on sampling episodes to estimate the policy gra-

dient. These sampled episodes introduce inherent variability due to the randomness in the environment and the policy itself. Therefore, to reduce the variance, a baseline b is introduced as follows:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[(G_t - b)\nabla_\theta \log \pi_\theta(a_t|s_t)] \tag{1.8}$$

Value-based methods are often used as a baseline, i.e., $b = V(s_t)$ which are known in the literature as Actor-Critic methods [171].

There is abundant literature regarding deep reinforcement learning. Readers may refer to the book of Hao et al. [71] for an in depth introduction.

1.3 End-to-end learning methods for the VRP

1.3.1 Machine learning for combinatorial optimization problems

Generally, there are two main reasons for using Machine Learning for combinatorial optimization [21]:

1. Expert knowledge is assumed about the optimization algorithm, and a fast approximation of heavy computations is desired. In this case, machine learning is used to learn to *imitate* the decision of the expert via supervised learning.

2. Expert knowledge is either not available or insufficient to derive a good decision-making algorithm. In that case, machine learning will help learn a strategy through exploring the decision space.

At the algorithmic level, the above-mentioned use cases are implemented in algorithm configuration learning, learning alongside optimization algorithms, and end-to-end learning.

Algorithm configuration is an important aspect of optimization algorithms, as many state-of-the-art metaheuristics have several parameters needed to be set before the resolution process begins. Often, expert knowledge is needed to set the parameters of an algorithm; they may also change based on instance features. Thus,

having an algorithm able to set them based on the problem characteristics can result in better quality candidate solutions for the problem [48]. Another aspect to consider is the choice of the algorithm to run to solve an instance. Machine learning models can be trained to predict, given an instance of a combinatorial optimization problem, which algorithm to use to output the best solution. For example, the *Learning to Improve* framework proposes to train a deep reinforcement learning agent to choose which local search operator to apply at each resolution step [118]. Learning alongside optimization generalizes the algorithm configuration so that it wraps the decision of the machine learning model inside the resolution loop along with the optimization algorithm. For example, a machine learning model can decide if it is advantageous to run a primal heuristic in a branch and bound algorithm [94].

Finally, end-to-end methods propose to learn a heuristic for solving optimization problems. A neural network is used as a general function approximation to output a solution for the optimization problem. It can be described as a mapping from the problem-parameters pair $(\mathcal{P}, \mathbb{R}^d)$ to the space of heuristics $\mathcal{H}$.

$$\begin{aligned} f : \mathcal{P} \times \mathbb{R}^d &\longrightarrow \mathcal{H} \\ (p, \theta) &\longmapsto f(p, \theta) = h \end{aligned}$$

The learned heuristic h can be either an existing one, or a completely unknown heuristic. In the first case, we aim at learning an existing heuristic when the solution's quality output by the heuristic is good, but it is time-consuming. The target heuristic is thus fixed, and we learn it by demonstrating, for each training instance, the desired solution to output. In the second case, we propose to search for a heuristic by trial-and-error, until convergence is reached. In this case, the target heuristic is not fixed, we move from a heuristic search h to another one h' using the solution quality achieved by the heuristic h on the instances as a signal to guide the neural network training.

End-to-end methods are intended for solving similar problems to the ones the deep neural network was trained on. They assume that the target instances come from the same distribution as the ones used to obtain the training dataset. For example, in routing problems, it may be problems that come from the same city

locations.

Let us note that all the previously mentioned algorithmic level implementations can be used one in cooperation with another. For instance, an end-to-end learned heuristic can be incorporated into a more general optimization algorithm. Neural Diving, best represents this cooperation: it proposes to learn variable assignments for a mixed integer programs from a set of collected data from SCIP solver [22]. This learned diving heuristic is then incorporated into the branch and bound algorithm [131].

In what follows, we propose to review end-to-end machine learning for the VRP. We will focus on the best known variants of the problem, along with the Traveling Salesman Problem. The main contributions are summarized in Tables 1.4, 1.5, 1.6, 1.7 (pages 76, 77, 78, 79).

1.3.2 Neural combinatorial optimization for the VRP

1.3.2.1 VRP as a supervised learning problem

In the supervised learning setting, an oracle provides the desired solutions to output, and a deep neural network learns to capture the patterns and relationships between the inputs (instances) and the outputs (solutions). The goal of training the deep neural networks is to be able to process instances unseen during training, by outputting good quality solutions. Let us note $(X_i, Y_i^*)_{i=1}^N$ a dataset of pairs of instances' features and their corresponding solutions obtained via an exact algorithm, or (meta)heuristic giving *"good quality"* solutions, such as HGS for the CVRP [181], Concorde for the TSP [6], or LHK3 for both problems [73]. Each instance X_i of the dataset is made of a set of coordinates of cities in the case of TSP, or clients, depots and the demands in the case of CVRP, i.e. $X_i = \{\mathbf{x}_j^i\}_{j=1}^n$, where

$$\mathbf{x}_j^i = \begin{cases} (a_j^i, b_j^i), & \text{where } a_j^i, b_j^i \in \mathbb{R} \text{ represent the 2D-coordinates of a city } j \text{ for TSP} \\ (a_j^i, b_j^i, q_j^i), & \text{where } a_j^i, b_j^i, q_j^i \in \mathbb{R} \text{ represent the 2D-coordinates of the depot} \\ & \text{or a client } j \text{ and its corresponding demand for CVRP} \end{cases} \tag{1.9}$$

The corresponding ground truth solution Y_i^* is represented by a sequence of visit orders, i.e. $Y_i^* = \{y_1^*, y_2^*, \ldots, y_T^*\}$. For example, in the case of TSP, y_1^* represents the city index that will be visited at first, and y_2^* is the index of the second city to visit, and so on. This sequence has a length T equal to the input's length, in the case of TSP, and is of different length due to the return to the depot, in the case of CVRP.

Formulating routing problems as supervised learning problems can be achieved either in an autoregressive setting or a non-autoregressive one.

Autoregressive setting. In this context, the solution is generated sequentially, similar to constructive heuristics that choose the next client to be visited at each step. The probability to output a solution Y_i for some instance X_i is given by the probability chain rule as follows:

$$P_\theta(Y|X) = \prod_{t=1}^{T} p_\theta(y_t|y_1, \ldots, y_{t-1}, X) \tag{1.10}$$

where $p_\theta(\cdot)$ is a deep neural network with parameters θ. The optimal parameters corresponding to the target heuristic θ^* are searched by maximizing the log probabilities of expected output Y_i^* for the instances X_i of the training set:

$$\theta^* = \arg\max_\theta \sum_{X_i, Y_i^*} \log P_\theta(Y_i^*|X_i) \tag{1.11}$$

"Pointer Network" (Ptr-Net) is the deep learning model that introduces this approach to solve combinatorial optimization problems [185]. This approach was tested on Convex Hull, Delaunay Triangulation and the Traveling Salesman Problem. It follows the Encoder-Decoder architecture with an attention mechanism to compute the probabilities. Two LSTMs are used as an encoder and a decoder, respectively. The LSTM encoder maps each city features x_j into a $d-$dimensional vector space $\mathbf{h}_j$. To do so, the hidden state output by the encoder, at each encoding step, is kept as the city's new representation. The decoder is responsible for computing, at each decoding step t, a query vector $\mathbf{q}_t$ that will be used in an additive attention mechanism to compute the probability of cities' selection, i.e.,

$$p_\theta(y_t = j|y_1, \ldots, y_{t-1}, X) = \frac{exp(u_t^j)}{\sum_{k=1}^{n} exp(u_t^k)}$$

where:

$$u_t^j = \mathbf{v}^\top \cdot tanh(\mathbf{W}_q \ \mathbf{q}_t + \mathbf{W}_e \ \mathbf{h}_j) \tag{1.12}$$

the query vector $\mathbf{q}_t$ is computed by passing into the decoding LSTM the features of the lastly selected city. Having access to ground-truth optimal solutions is often impossible or time-consuming. To overcome this, Objective-based training is introduced [126]. The deep neural network is still trained in a supervised way, but using instances with a heuristically generated solution using, for example, the Nearest Neighbor heuristic. Instead of directly training the deep neural network to mimic the heuristic, the solution is retained, if and only if it is better than the one that is output by Ptr-Net. Thus, if the solution found by the neural network Y is better than the ground-truth heuristic solution Y^*, in terms of objective function, then, it is not necessary to adjust the weights of the neural network because in this case, it will learn for a worse quality solution than the one it found [126], i.e.,

$$\theta^* = \arg\max_\theta \sum_{X_i, Y_i^*} \mathbb{1}_{f(Y_i^*)<f(Y_i)} \log P_\theta(Y_i^*|X_i) \tag{1.13}$$

with $f(\cdot)$ being the objective function and $\mathbb{1}_{f(Y_i^*)<f(Y_i)} = \begin{cases} 1 & if \ f(Y_i^*) < f(Y_i) \\ 0 & otherwise \end{cases}$

Ptr-Net with Objective-based training was later used to tackle a Pickup and Delivery problem (Ride Hailing). A Large Neighborhood Search (LNS) metaheuristic is used to generate the approximate solutions for training. The resulting trained model is then used as an insertion heuristic inside the LNS. The motivation behind this hybridization is to take advantage of the past runs on similar problems. The trained model turned out to be better than a handcrafted greedy construction heuristic, and improved the quality of LNS compared with a handcrafted insertion procedure [172]. Ptr-Net uses neural network weights as a means of storing relevant information, but as the weights get updated, information is overwritten. To avoid this, an approach that uses an explicit memory module was proposed for TSP and the CVRP [177]. This approach relies on Differentiable Neural Com-

puters (DNCs), which explicitly and intentionally (over)write information to an external memory. The DNC can be viewed as a differentiable CPU having access to a differentiable RAM [69]. The model stores edge information, then performs a read operation to retrieve the edges that are in the solution. While this approach turns out to be effective to solve TSP and CVRP, it is hard to train, since the number of edges grows quadratically with the number of vertices. The approach required training using curriculum learning, by gradually increasing the size of the instances, until it reaches the size of the desired problem to tackle.

A recent method proposed by Li et al. [113] proposes to tackle large-scale CVRP instances ranging from 500 to 3000 clients. Their approach proposes to train a Transformer model, via supervised learning, to generate CVRP sub-problems and to pass them to black-box solvers. The algorithm starts from a feasible solution, and defines the sub-problems as a set of routes in the current solution. The sub-problems are built by first selecting a route and computing its centroid, then the k nearest routes are added. The nearest routes are determined by the Euclidean distance between the centroid of each route and the initially selected route's centroid. The sub-problem selection is then formulated as a regression problem, where a Transformer model determines the cost of passing the sub-problem into a solver. In the training dataset, this cost is already determined for each sub-problem using, for example, the LKH3 and HGS-CVRP algorithms. At test, the sub-problem which yields the best improvement in terms of cost is selected. The method is competitive with LKH3 while being faster to execute. This cooperation between learning methods and handcrafted methods proves to be more effective in dealing with large instances, up to 3000 nodes, which pure learning-based methods still struggle to tackle.

Non-autoregressive setting. In this case, the deep neural network is designed to output the solution in a one-shot fashion, by producing a probabilistic heatmap over the adjacency matrix. Table 1.3 shows an example of such a heatmap for a TSP instance with 4 cities, which corresponds to the probability that an edge is part of the optimal solution. For instance, there is a probability of 0.3 that the edge (1,3) is part of the optimal solution.

This setting was introduced for the TSP in the work of Joshi et al. [89]. Their

	1	2	3	4
1	0.00	0.20	0.30	0.50
2	0.10	0.00	0.50	0.40
3	0.30	0.40	0.00	0.30
4	0.40	0.50	0.20	0.00

Table 1.3: Probability heatmap for TSP edges

model is based on a stack of Graph ConvNet layers, which computes new nodes and edges representations for each instance. The node features used are the $2D$-coordinates, while the edges features correspond to the distance between each city at the ends of an edge. The edges embeddings are then converted into a probabilistic heatmap using a Multi-Layer Perceptron. This approach, while effective in practice, can become intractable to train for large instances, as for a batch of B instances of n cities, $B \times n \times n$ heatmaps needs to be computed, which would make the space complexity of this approach intractable for large values of n. Also, knowing that numerous instances are needed to train the model (in the order of one million), obtaining the optimal solutions for large instances becomes a bottleneck. This model was later on trained on CVRP instances with ground-truth generated by LKH3 [99]. A similar approach, which uses a different graph neural network encoder and more advanced search strategies (see subsection 1.3.3) was proposed to tackle large TSP instances [58].

While supervised learning turned out to be able to learn heuristics for solving VRPs, applying it to real-world instances is still challenging. First, it requires labeled data, which ideally would be ground-truth optimal solutions. While Objective-based training showed that it is possible to get ground-truth solution from heuristics and train the neural network towards better solutions, it is not guaranteed that the learned heuristic is of better quality. For non-autoregressive models, while they tend to output better solutions than autoregressive models, they require a pre-defined fixed size output for the model, which means that either a model needs to be trained for each instances' size (e.g., TSP with 100 nodes), or advanced exploitation techniques such as graph sampling, heatmap merging, Monte Carlo Tree Search (MCTS) are required to achieve good results [58].

1.3.2.2 VRP as a reinforcement learning problem

In combinatorial optimization problems, we have access to an objective function which can quantify the quality of a solution. Luckily, reinforcement learning can exploit this signal as a reward or a cost to provide feedback to the deep neural network models, and to guide them towards the parameters that output good quality solutions. The approach, known as Neural Combinatorial Optimization (NCO), was introduced by Bello et al. [20] as a *"framework to tackle combinatorial optimization problems using reinforcement learning and neural networks"*. The advantage of this approach is that it does not require ground-truth labels, as in supervised learning. This is a further step towards the automatic definition of heuristic methods for difficult $\mathcal{NP}$-hard problems. In this case, the solution space includes all possible, even unknown, heuristic search algorithms, which is at the same time, an advantage, and a disadvantage. If trained well, reinforcement learning-based policies are a great opportunity to discover efficient unknown heuristics for difficult optimization problems. The risk, in this case, is that the policy search process gets stuck into a local optimum, which yields a worse heuristic method than a manually defined one. As for handcrafted heuristic methods, the framework can be used to define constructive methods or improvement methods.

Construction-based neural combinatorial optimization. These methods start from an empty solution and iteratively adds nodes to it until a stop criterion is met. For the CVRP and TSP, the criterion is the visit of all clients/cities. This iterative process corresponds to a sequential decision-making problem. Thus, it is possible to model the problem using a finite MDP, and therefore use reinforcement learning to solve the problem. Following an MDP formulation, states, actions, and reward function are defined as follows:

- **State (s_t):** it reflects the evolution of the solution process. Thus, it corresponds to the instance features as defined in Equation 1.9 (page 55) and an empty solution at $t = 0$, or the partial solution under construction for $t > 0$.

- **Action (a_t):** it corresponds to the choice of the next node (city/client) to add to the partial solution.

- **Reward (r_t):** it is often deterministic and reflects the change in the cost function after taking action a_t and transitioning to a new state s_t. It is defined as the total distance traveled following the partial solution until the decision step t. When the termination state is reached, the cumulative reward corresponds to the objective function of the routing problem.

Let us note that, in the case of deterministic routing problems, the dynamics of the MDP are also deterministic, i.e., the probability of transitioning to a state s' while being in a state s and taking the action a is always equal to 1; $P(s_{t+1} = s' | s_t = s, a_t = a) = 1$.

In Neural Combinatorial Optimization for routing problems, the goal is to learn the parameters of a stochastic policy $P_\theta(\cdot|X)$, represented as a deep neural network, which, given an instance $X = \{x_j\}_{j=1}^n$ of the problem, assigns high probability to good quality solutions, while it lowers the probability of bad quality ones. This probability is defined using the probability chain rule, as in equation 1.10 (page 56). For simplicity, we will first detail the framework, in the case of the TSP. In this case, a solution of a TSP instance X represents a visit order of the cities in the form of a permutation $Y = \{y_1, y_2, \dots, y_n\}$, where y_t is the index of the city that is selected at step t. Thus, the training objective is the minimization of the expected TSP tour length of the solutions Y sampled from the model for a given instance X, i.e. $J(\theta, X) = \mathbb{E}_{Y \sim p_\theta(\cdot|X)}[L(Y, X)]$, with $L(Y, X) = \sum_{t=1}^{n-1} \|x_{y_t} - x_{y_{t+1}}\| + \|x_{y_n} - x_{y_1}\|$.

Generally, the model is trained to tackle a class of instances of the same problem, which means that the instances come from the same distribution $\mathcal{D}$. The training objective is, thus, formulated as the minimization of the expected tour lengths of the solutions Y sampled from the deep neural network, i.e. $J(\theta) = \mathbb{E}_{X \sim \mathcal{D}}[J(\theta, X)]$. To do so, we resort to the gradient descent algorithm to find the parameters θ of the model. The gradient is given by the REINFORCE with baseline algorithm: $\nabla_\theta J(\theta) = \mathbb{E}_{X \sim \mathcal{D}, Y \sim P_\theta(\cdot|X)}\Big[\Big(L(Y, X) - b(X)\Big)\nabla_\theta \log P_\theta(Y|X)\Big]$, which itself is estimated using the Monte Carlo sampling (Algorithm 3 depicts the whole training procedure):

$$\nabla_\theta J(\theta) \approx \frac{1}{B} \sum_{i=1}^{B} \Big[\Big(L(Y_i, X_i) - b(X_i)\Big)\nabla_\theta \log P_\theta(Y_i|X_i)\Big]$$

Algorithm 3: REINFORCE with critic baseline for the TSP (adapted from Bello et al. [20])

Inputs: policy network P_θ, baseline network b_ϕ, number of epochs E, batch size B, number of instances K, number of cities n
$T \leftarrow \frac{K}{B}$
for $e \leftarrow 1$ *to* E **do** // train for E epochs
  for $t \leftarrow 1$ *to* T **do** // loop over the T instance batches
    $X_i \leftarrow \text{getInstance}(n), \quad \forall i \in \{1, ..., B\}$
    $Y_i \leftarrow \text{SampleSolution}(X_i, p_\theta), \ \forall i \in \{1, ..., B\}$
    $b_i \leftarrow b_\phi(X_i), \ \forall i \in \{1, ..., B\}$
    $L_i \leftarrow L(X_i, Y_i) \ \forall i \in \{1, ..., B\}$
    // Compute the loss and update the neural network parameters
    $\nabla_\theta J_\theta \leftarrow \frac{1}{B} \sum_{i=1}^{B} (L_i - b_i) \nabla_\theta \log P_\theta(Y_i | X_i)$
    $\mathcal{L}_\phi \leftarrow \frac{1}{B} \sum_{i=1}^{B} \|b_i - L_i\|$
    $\theta \leftarrow \text{Adam}(\theta, \nabla_\theta J_\theta)$
    $\phi \leftarrow \text{Adam}(\phi, \nabla_\phi \mathcal{L}_\phi)$
  end
end

Similar to Ptr-Net, NCO follows the encoder-decoder approach. Bello et al. [20] uses a LSTM encoder and decoder with an additive attention and mask mechanism for computing the probability of selecting a city for the TSP. The mask mechanism intervenes when computing the attention scores. This mechanism is problem-dependent, as it faithfully transcribes the problem constraints. For example, in the case of TSP, the already visited cities are masked, by forcing the attention score value of the city to $-\infty$, thus when using the softmax function, its corresponding probability would be zero. Moreover, the attention scores are clipped using a parameter C. This parameter controls the entropy of the model. The higher is the value of C, the more confident the model is.

$$u_j^t = \begin{cases} C \cdot tanh\left(\mathbf{v}^\top \ tanh(\mathbf{W_q} \ \mathbf{q_t} + \mathbf{W_e} \ \mathbf{h_j})\right), & j \notin \{y_{t'}\}_{t'<t} \\ -\infty & \text{otherwise} \end{cases} \tag{1.14}$$

with $\mathbf{h_j}$ being the embedding of the node x_j resulting from the encoding phase, and $\mathbf{q_t}$ is the query vector resulting from the *glimpse* mechanism, and $y_{t'}(t' < t)$ are the already visited cities in the previous steps.

The glimpse mechanism is used to aggregate the contributions of different part of the input to compute a query vector. In practice, this mechanism uses the attention over the city embeddings $\{\mathbf{h_j}\}_{j=1}^{n}$ and the decoder's output d_t vector followed by a softmax to compute attention weights $\{\alpha_i^t\}_{i=1}^{n}$. These weights are then used to compute a query vector q_t as a weighted convex sum of the cities embeddings.

$$\mathbf{q_t} = \sum_{i=1}^{n} \alpha_i^t \mathbf{h_i}$$

The baseline is an estimation of the value of the tour length. It is computed by a deep neural network, named *critic* network $b_\phi(\cdot)$. Bello et al. [20] uses a LSTM encoder and a MLP with two hidden layers as critic network. It takes as input the instance X and outputs a single scalar which estimates the optimal tour length. This deep neural network is trained to minimize the mean squared error objective between its predictions $b_\phi(X)$ and the actual tour lengths sampled by the policy $p_\theta(\cdot|X)$, i.e.,

$$\mathcal{L}_\phi = \frac{1}{B} \sum_{i=1}^{B} \|b_\phi(X_i) - L(Y_i, X_i)\|, \quad \text{with } Y_i \sim P_\theta(\cdot|X)$$

The model was trained on instances of size 20, 50, 100 and showed competitive results against handcrafted constructive methods, such as Christofides and OR-Tools.

Nazari et al. [132] adapted the NCO framework for the CVRP and SDVRP. For this, the authors divided the instance features into static and dynamic ones. The static features being the coordinates of the clients and the dynamic features being the demands, since they change across the resolution process (a selected client sees its demand become zero to reflect that it is completely fulfilled).

They defined an encoder-decoder model which consists of a single layer MLP encoder without an activation function[14] and a LSTM decoder with additive atten-

[14] this is equivalent to performing a linear transformation of the input.

tion. Using a MLP for encoding removes the bias induced by the LSTM encoder on the order of inputs. This allows them to encode the instance as a set of points instead of a sequence. They, however, kept the LSTM decoder since, the resulting candidate solution can be viewed as a sequence of visit order. The LSTM takes only the static features of the last selected client as input, as using both static and dynamic ones does not show any improvement. Omitting them has the advantage of reducing the number of parameters to learn for the deep neural network model, while not degrading performance. Similar to Bello et al. [20], a glimpse mechanism is used to compute the query vector using the decoder's output, and the static and dynamic embeddings for the clients.

They further extended the mask mechanism so that the following attention scores are masked at each step t for:

- Clients with null demands;
- The depot, if it was visited at the step $t-1$, to prevent the model from being stuck at the depot. It becomes eligible again at $t+1$;
- Clients with demands greater than the vehicle's capacity, in the CVRP case.

The model was trained on instances of 10, 20, 50, 100 clients using REINFORCE with Critic baseline. It yields better solutions compared with Clarke and Wright savings heuristic, the Sweep algorithm, and OR-Tools.

In the work of Nazari et al. [132], only VRPs with homogeneous vehicle capacities were tackled. More precisely, there is no restriction on the number of available vehicles, only the total distance traveled should be minimized. This work has been extended for the heterogeneous case [180]. In addition to the clients' features (coordinates and demands), the state is represented by vehicles' features (coordinates and remaining capacity). They modeled the problem as a multi-agent one; a policy is trained for each vehicle (agent), in a centralized and cooperative way. This is achieved by making the agents share the same state, and decide in a sequential and alternating way. They extended the deep neural network model of Nazari et al. [132] by introducing vehicle features, and used Advantage Actor Critic (A2C) to train a policy for each vehicle [129]. The algorithm was experimented on HCVRP with 10, 20, 50 and 80 clients and 3 vehicles with different capacities. The results

indicate that the model is competitive with the Sweep and Clarke and Wright savings heuristics, but it does not beat OR-Tools solver. The authors argue that training the agents in an alternating way introduces a bias by assuming that the optimal solution is achieved by alternating between the decisions for each agent. One possible limitation of their framework is the need of training different policies for each vehicle, but no further investigation was made available by testing the model on a different number of vehicles.

Following the NCO framework, Kool et al. [100] introduced in 2019 the Attention Model (AM), a deep neural network architecture fully based on the attention mechanism for tackling several routing problems: TSP, CVRP, SDVRP, Orienteering Problem (OP), Prize Collecting TSP (PCTSP) and Stochastic PCTSP. They also introduced a new baseline which speeds up learning. Their encoder uses two MLPs: one for computing the clients' embeddings and another one for the depot embedding (which they consider as a different node). Then, a Transformer encoder with 3 Transformer blocks is used to compute a new representation of the nodes (clients + depot) and a graph representation by using a mean pooling layer. Their decoder uses a glimpse mechanism to produce a query vector, by using the graph representation, the lastly visited client and the vehicle's remaining capacity. Their glimpse is based on a Multi-Head Attention. The query vector and the clients/depot embeddings are then used in a scaled dot-product attention with a masking mechanism and a softmax function to produce a probability distribution for the next node selection. They trained their model using the REINFORCE algorithm with Greedy Rollout baseline. This new baseline uses a copy of the learning network (rollout policy), which is updated less often. This copy produces a solution greedily, by choosing at each step the node with the highest probability. The baseline is updated by comparing the current learned policy with the rollout policy on a set of instances unseen during training. If the improvement is significant according to a paired t-test with a threshold of (5%), then the rollout policy is updated. Their computational experiments indicated that this baseline yields faster convergence than a critic baseline. However, it requires additional time, since it constitutes an additional forwards pass. This model showed better results than handcrafted heuristics and Nazari et al.'s model, on a wide range of routing problems. It rapidly became the state-of-the art model, and serves as

a baseline for both comparison and future research directions. For example, the work of Zhao et al. [207] extends the model of Nazari et al. [132] and uses Kool et al. [100] training algorithm (REINFORCE with Greedy Rollout baseline) for training, on the CVRPTW. To do so, they extended the mask to include the time windows constraints. Furthermore, the authors suggested that the resulting solution of the model would make a good warm start solution for metaheuristics. They hybridized their model with Guided Local Search of OR-Tools and LNS. Their findings show that better results are achieved when using their model's solution as a starting point, than when using a handcrafted heuristic's solution. Other applications of this learning-based models include the Electric Vehicle Routing Problem with Time Windows (EVRPTW) [114] and the Orienteering Problem with Time Windows (OPTW)[60].

Other works try to improve over the Attention Model (AM), such as Policy Optimization with Multiple Optima (POMO) algorithm [103]. POMO introduces a novel way of training NCO models for routing problems. It exploits the symmetries in the representation of the instance's solution. For example, in a TSP instance with 5 nodes, if the sequence $2 \rightarrow 5 \rightarrow 3 \rightarrow 1 \rightarrow 4 \rightarrow 2$ is a solution, then the following sequence is also an optimal solution $3 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 3$. The goal of training with POMO is to teach the deep neural network these kinds of solution symmetries. For this, instead of sampling a single candidate solution in the training phase, η candidate solutions are sampled. This would correspond to replacing the line 6 in algorithm 3 by:

$$Y_i^j \leftarrow \text{SampleSolution}(X_i, P_\theta, \eta), \quad \forall j \in \{1, ..., \eta\} \forall i \in \{1, ..., B\}$$

POMO also uses a simpler shared baseline among the candidate solutions which corresponds to the average solution length over all the sample solutions, i.e., $b(X_i) = \frac{1}{\eta} \sum_{j=1}^{\eta} L(Y_i^j, X_i)$. Although these symmetries are not the same for the CVRP because not every client node can be the first node to visit in route, the same policy was applied to this problem, and showed strict improvements in the solution quality. Finally, the authors suggest using more Transformer blocks (6) in the Transformer encoder to improve the model's performances.

The Joint Attention Model for Parallel Route-Construction (JAMPR) proposes

to modify the Attention Model to tackle CVRP and CVRP with soft, partly soft and hard Time Windows [51]. They argue that the sequential construction of single routes yields poor performance because no information about the previously constructed routes arrives at the current route under construction, but only information about the lastly selected node. The difficulty of this construction approach augments as more constrained problems are tackled (e.g., CVRPTW). For this, they extended the AM with a Tour encoder and a Vehicle encoder. The learned policy's action space is made of pairs of the uncompleted routes and the remaining clients.

Inspired by the Bellman's Principle of Optimality, some research directions argue that, once a node is chosen at some resolution step, it is no longer needed to be part of the sub-problem induced by the remaining non-routed nodes [144, 196]. To do so, each time a node is selected for a visit, it is removed from the instance. Nodes which do not satisfy the capacity constraint are also removed from the sub-problem. A dynamic embedding mechanism is used on the remaining nodes, i.e., they are re-embedded using the encoder. Embedding at each time step allows having a more accurate representation of the remaining sub-problem to solve. This strategy improved the accuracy of Ptr-Net [196] and AM [144, 196]. However, re-embedding the nodes is computationally costly, which induces longer training times. For example, for a TSP instance with n nodes, a transformer model has a complexity of $O(n^2)$, and calling the encoder n times, would result on a complexity of $O(n^3)$. In practice, this limits the use of this technique. Alternately, to consider the evolution of the resolution process, it is possible to inject this information into the context vector. This direction was followed by Xu et al. [198]. In the original Attention Model, the context embedding is given by a concatenation of the graph representation, the lastly visited client and the vehicle's remaining capacity, for the CVRP. The fact that the graph representation is static gives poor information about the evolution of the resolution process. The information of the last visited client and the remaining vehicle's capacity is not sufficient. In the Multiple Relational Attention Model of Xu et al. [198], the context embedding considers the resolution process. To do so, they compute an embedding of the visited nodes and another of the unvisited nodes, at each resolution step. These two embeddings replace the static graph representation on the context embedding

computation. This adjustment improves the results.

Another way to lead to fast training convergence is to diversify the outputs of the deep neural network models. For this, instead of learning one policy, multiple policies can be trained simultaneously, which increases the chances of finding good quality solutions. This is what the Multi-Decoder Attention Model (MDAM) proposes [197]. Inspired by the Multi-Head Attention, MDAM uses M decoders with identical structures and different parameters. Each decoder outputs a different candidate solution. The encoder and the decoders follow the designed architecture of AM, except that the last encoding layer uses a dynamic encoding layer similar to Xin et al. [196] model. The model is trained using the REINFORCE algorithm. To ensure diversification, a Kullback-Leibler (KL) divergence between each pair of the output probability distributions from the multiple decoders of MDAM is maximized. For a given solution Y of an instance X, the KL divergence is defined as:

$$D_{KL}(\theta_1, \ldots, \theta_M) = \sum_{i=1}^{M} \sum_{j=1}^{M} \sum_{t=0}^{|Y|} p_{\theta^i}(y_t|y_0, \ldots, y_{t-1}, X) \log \frac{p_{\theta^i}(y_t|y_0, \ldots, y_{t-1}, X)}{p_{\theta^j}(y_t|y_0, \ldots, y_{t-1}, X)}$$

This approach improves the performance of the Attention Model for routing problems up to 100 nodes. An interesting approach suggests combining reinforcement learning and supervised learning to train an encoder-decoder model for the CVRP [46]. This joint learning approach uses a GCN encoder with edges features, and two decoders. The first decoder is a GRU recurrent cell with an additive attention mechanism, as defined by Bello et al. [20]. The second one is a MLP, which output a probabilistic heatmap similar to the supervised non-autoregressive work in [89]. The solution sequence output by the first decoder is converted to an adjacency matrix, which is used as the supervised label for the second decoder. The whole neural network is trained to minimize the sum of REINFORCE loss and the binary cross-entropy loss.

As we can see, most of the RL-based methods for routing problems are based on direct policy search via policy gradient methods, with variants of REINFORCE as a training algorithm. A few other works consider deep learning value-based methods, such as Deep Q-learning. It is more sample efficient than policy-based

methods, because of the experience replay buffer. The replay buffer stores tuples of the form of (s_t, a_t, r_t, s_{t+1}) in a memory, and they are later used to train the deep neural network. S2V-DQN is the first method of this kind that has been applied to tackle the TSP. It combines the representation power of graph neural networks (STRUC2VEC [154]) and the n-step Deep Q-learning algorithm for training. STRUC2VEC is used to make nodes representation, by aggregating embeddings of a fixed number of nearest neighbors of each node. These nodes' embeddings are then used in a sum pooling layer to form the graph embedding. The nodes embeddings and the graph embedding are used in an attention-like mechanism to compute a Q-value for each node. The action selection is performed by taking the node with the highest Q-value. Another work tackles both static and dynamic[15] variants of the CVRP by using a Graph Neural Network trained with Deep Q-learning [135]. Their deep Q-network was trained in an offline setting, using a simulator. While the performance falls short against classic handcrafted heuristics on the static CVRP, the model showed better performance on the dynamic CVRP. A third work considers an adaptation of the Attention Model [100] to the deep Q-learning setting to tackle the CVRP and the Multi-depot VRP (MDVRP) [16]. The model, named RP-DQN, introduces a new way of encoding dynamic node embeddings by injecting information about whether a node (i) has already been inserted in the tour, (ii) cannot currently be inserted due to capacity constraints, (iii) represents the currently served client or, in the multi-depot case, represents the current depot. They trained the neural network via Deep Q-learning. Their algorithm obtained better performance than the policy-based Attention Model on both CVRP and MDVRP, while being more sample efficient. In OmegaZero [186], a Deep Q-network based on GAT encoder and GRU decoder is introduced. The model is inspired by AlphaZero and is trained via self-play, i.e. the model is trained to compete against its previous versions.

With regard to instance features, most of the approaches propose to input directly the raw features, which consist of the coordinates and the demands, for the CVRP. Carefully designed features are essential in machine learning, even though deep learning has a strong ability for automatic feature extraction. For example, to enhance the representation power of the graph neural networks, the

[15] In the dynamic CVRP, the clients requests appear as the resolution process occurs.

distance between a pair of nodes can be injected as an edge feature [93, 46, 111, 114]. Instance features are an important research line in order to improve the performance of the models.

Improvement-based neural combinatorial optimization. Similar to handcrafted improvement heuristics and metaheuristics, these methods start from an initial solution, and try to improve it. To do so, these methods integrate a learning component. These types of methods tend to generally give solutions with better quality than construction neural combinatorial optimization methods. This is because they perform more exploration of the solution space, especially in inference.Analogously to constructive approaches, most of the improvement-based neural methods use reinforcement learning. Following MDP formulation, states, actions, and reward function are defined as following:

- **State (s_t):** corresponds to the instance features and the incumbent solution from previous states. At $t = 0$, an initial solution is generated via a simple handcrafted heuristic such as nearest neighbor.

- **Action (a_t):** is dependent on the approach being implemented. It can be, for example, the selection of the node to move in the solution.

- **Reward (r_t):** reflects the amount of improvement (or deterioration) achieved by the policy while performing the action a_t. It is computed as the difference between the objective function of the solution at s_t and the incumbent.

We find, in the literature, two frameworks that integrate improvement neural methods. The Large Neighborhood Search (LNS) and another one that bears similarities with Iterated Local Search (ILS).

The Large Neighborhood Search (LNS) framework. LNS is a metaheuristic that iterates between destruction and repair operators on a solution. The destruction operator removes clients from the solution, while the insertion operator inserts them back into routes such as the obtained solution is better. The first work reported of this kind has been applied to a ride hailing problem [172], which we

discussed earlier in the supervised learning subsection 1.3.2.1. Concurrently, Hottung and Tierney developed Neural Large Neighborhood Search (NLNS), which integrates a learned heuristic inside the Large Neighborhood Search metaheuristic [80]. In NLNS, the destruction procedures are handcrafted, while the insertion one is carefully designed to be integrated inside the metaheuristic. Integrating learned insertion heuristics inside a LNS framework has advantages over leaned construction heuristics. For instance, the neural network produces shorter sequences, since only a part of the removed clients needs to be re-routed. This allowed NLNS to tackle instances up to 300 clients. The effectiveness of the metaheuristic was demonstrated on two destroy operators: a client-based destroy, which removes clients closest to a randomly selected point and a tour-based destroy, which removes the tours that are close to a randomly selected point. This results in an incomplete solution where either a client is not routed at all, or only partially routed, with one end not connected to another client or the depot. The non-routed and partially routed clients serve as input to the deep neural network model. The designed deep neural network consists solely of MLPs. In the first step, a non-routed client is selected to be a reference client, and will serve as a query vector. The reference client is passed into a MLP to generate a reference embedding. The rest of the clients are passed into another MLP to obtain embeddings. The reference and clients embeddings are used in a first additive attention module similar to the glimpse mechanism [20], to generate a context vector. This vector is used to encode information on relevant inputs and is concatenated with the reference embedding to compute a query vector. The query and the clients' embeddings are then used in an additive attention to compute a probability distribution over the non-routed clients. The model is trained using a cost signal representing the cost of repairing the initial solution (the difference of tour length between the destroyed solution and the repaired solution). The REINFORCE with critic baseline algorithm is used to train the model [80]: the critic is a MLP that estimates the cost of repairing a solution. A model is trained for the different destroy operators. The results indicate that the NLNS performs better than a LNS with handcrafted insertion procedure.

NeuLNS is another Large Neighborhood Search-based learning algorithm, designed to tackle the CVRP and CVRP with Time Windows [61]. Contrary to NLNS

[80], NeuLNS learns to output which clients are removed from the solution, and in which order they are considered for insertion. They follow the encoder-decoder approach of Ptr-Net. The encoder is much more sophisticated as it integrates the graph structure of the solution. It uses an extension of Graph Attention Network with edges embeddings (EGATE) that includes information about the solution's arcs. The decoder uses a GRU recurrent cell to compute the query vector. Along with the nodes embeddings, they are passed to an attention mechanism to select the clients that will be removed from the tour. The selected client is then injected to the GRU to compute the new query. The fact that they are passed through a GRU reshapes the set of clients into an ordered list, which defines the order in which they will be re-inserted into the solution. The insertion follows a least-cost principle to yield a minimum cost solution. Thus, in a way, the insertion methods of NeuLNS can be considered as a mix between handcrafted and learning strategy, since the model gives only the order in which the removed clients are processed and not directly the position where they are inserted. The method was able to process medium-scale instances up to 400 clients, however, direct comparison with other approaches is not possible, since the method was evaluated using a different protocol.

Inspired by the recent SISR metaheuristic [33], a Large Neighborhood Search based on the removal of a cluster of clients has been proposed by Chen et al. [29]. Their method named Dynamic Partial Removal (DPR) uses a deep neural network model to determine an anchor node among the clients, which will serve as the basis for the removal of other clients. The model also outputs two other scalars. They are used to parameterize a beta distribution, which serves to determine the percentage of clients that will be removed from each route. The deep neural network (HRGCN) consists of an encoder made of a stack of GCN layers with skip connections, and a GRU decoder with an attention mechanism to compute the probability of selecting a node as an anchor node, similar to Ptr-Net. The removed clients are randomly shuffled and a least-cost based insertion procedure is used to create a new solution, which serves as input in the next iteration. The neural network is trained using a Proximal Policy Optimization algorithm. Empirical evaluation on the CVRPTW showed that the method is effective and competitive with the handcrafted LNS and SISR, on instances up to 800 clients.

1.3.2.2.1 The Iterated Local Search (ILS) framework. Methods of this kind uses a deep neural network to select the regions where a local search operator will be applied. This process is iterated in a number of predefined steps, starting from the previously found solution. The Learning to Delegate [113] method discussed in the supervised learning subsection can also be considered belonging to this framework.

NeuRewriter can be considered as the first method of this kind. It learns a two-parts policy: a region-picking and a rule picking policy [30]. For this, an initial solution is generated, using the nearest neighbor heuristic. Then, a region set composed of the solution's routes is defined. The region-picking policy selects one region, then the intra-route RELOCATE operator is used to move one client after another. The client to which the operator is applied is selected using a rule-picking policy, which is similar to Ptr-Net. The two policies are trained together, using a combination of Q-Learning and Actor-Critic Algorithm for the region-picking and rule picking policy respectively. Another line of research considers learning to select the clients to which pairwise operators, such as 2-OPT or SWAP, are applied. When designed in a handcrafted way, these operators run in $O(n^2)$, with n being the number of nodes. The goal of the deep neural network is to define the pair of nodes on which these operators will be applied given the actual solutions characteristics, which would reduce the complexity to $O(1)$. Wu et al. [195] designed a strategy of this kind. At each step, a solution is selected from the neighborhood of the current solution by performing a local search using a pairwise operator, with the pair of clients selected by a deep neural network. For this, the authors used a Transformer model along with a positional embedding function to obtain a solution embedding. The role of the positional embedding is to encode the position of each client in the current solution. Without this element, the encoding process would be similar to the Graph Neural Network encoding, resulting in a permutation equivariant model. A graph embedding is then computed using a max pooling layer and passed into a MLP. Each client embedding is passed into a MLP and is added up to the graph embedding. These embeddings are used in a dot-product similar to the self-attention mechanism to compute a compatibility matrix that is transformed into a probability matrix using the softmax function, which reflects the probability that a pair of clients will be selected for the pairwise operation. The algorithm is

trained using n-step actor-critic, with a Transformer-based critic. The empirical study indicated that the approach gives better results when used with the 2-OPT operator than with SWAP and RELOCATE. The approach is compared with its handcrafted counterpart, where the pairs of nodes to which the operator OP is applied is determined by either *first-improvement* or *best-improvement* policy, with a random restart when stuck in a local minimum. The results show that the learned method outperforms the handcrafted decision rules in less or equal iterations.

Ma et al. [121] follows the same strategy as Wu et al. [195], but uses a different model. Their method, Dual-Aspect Collaborative Transformer (DACT), learns nodes and positional embeddings separately. Then, it uses a Transformer encoder to compute a new representation, where the self-attention of each type of embedding uses the other type of embedding to compute the new representation. They introduce a new positional encoding which considers the cycling aspects in the routes of a CVRP solution. The decoder computes a compatibility matrix for each type of embedding, following the same strategy of Wu et al. [195], and combines them using a MLP and the softmax function to obtain the probability matrix. The model is trained using a Proximal Policy Optimization algorithm. Furthermore, they used curriculum learning during training, where the initial solution used as a starting solution are ordered in a decreasing order in terms of solution quality. Their results show that their model improves over the results of Wu et al. [195] in both CVRP and TSP, while being competitive and faster to execute than LKH3 heuristic.

Finally, a recent method, using end-to-end learning components, is introduced in [95]. It consists of a hierarchical solving protocol that learns collaborative policies (LCP). This method is a hybrid between a constructive policy (the seeder) and an improvement policy (the reviser). The seeder can be any constructive policy, such as the Attention Model. It constructs various initial candidate solutions that will be passed to the reviser policy. Then, the reviser decomposes each candidate solution into a finite number of disjoint segments of the same length, and updates them by performing a local search on each segment. The local search is performed by a deep neural network, similar to constructive models such as Attention Model, which constructs several candidate segments and returns the best, in terms of total travel length. The improved segments are then fused to form the complete

solution. This method showed that the handcrafted local search operators can be successfully replaced by a deep neural network model to determine the neighboring solution to select.

1.3.3 Search strategies

The purpose of training a model is to discover a policy that can tackle instances of the considered routing problem unseen during training. Search strategies, or inference strategies, are an important line of research, since they define the way the model is exploited to output a solution. They influence the final performance of the model. In what follows, we denote X the routing problem instance, $Y = \{y_1, \ldots, y_T\}$ the solution sequence, $p_\theta(\cdot)$ the learned policy, and $Q(s_t, y_t)$ the Q-value of a given state-action pair. Furthermore, we will focus on search strategies for construction-based approaches, since there is abundant literature on the subject.

Greedy search. Greedy search generates a solution by taking, at each step, the client with the highest score. In the case of supervised learning [172], or policy-based reinforcement learning [20, 132, 100], i.e.,

$$y_t = \arg\max_y p_\theta(y|y_1, \ldots, y_{t-1}, X), \quad \forall t \in \{1, \ldots, T\}$$

In value-based reinforcement learning, this corresponds to the node with the highest Q-value at resolution step [93, 16].

$$y_t = \arg\max_y Q(s_t, y; \theta), \quad \forall t \in \{1, \ldots, T\}$$

Figure 1.15 depicts an example of the first three steps of solving a TSP instance with five cities under the NCO framework, using greedy search and random search based on a uniform distribution. The deep neural network (DNN) outputs at each step a probability distribution to extend the partial tour under construction by a given city. In the first step, the state s_0 contains information about the instance in the form of the city coordinates and the partial tour is empty. Using the greedy

Approach	Problems	Supervised learning	RL method	Type	Model	Instances size	Training dataset size	Search strategy
Ptr-Net [185]	TSP	Autoregressive	/	Construction	LSTM encoder and LSTM decoder	5, 10, 20,50	1M	Beam Search
Ptr-Net with Objective-based learning [126]	TSP	Autoregressive	/	Construction	LSTM encoder and LSTM decoder	20	10K	Beam Search
Memory augmented neural networks[177]	TSP, CVRP	Autoregressive	/	Construction	DNC	10-30	/	/
GCN [89]	TSP	Non-autoregressive	/	Construction	Graph ConvNets with MLP	20, 50, 100	1M	Greedy Search, Beam Search
DPDP [99]	TSP, CVRP	Non-autoregressive	/	Construction	Graph ConvNets with MLP	100	1M	Dynamic Programming
Neural Network Based LNS [172]	RHP	Autoregressive	/	Construction & Improvement	LSTM encoder and LSTM decoder	10, 20, 30, 40	1000	Greedy Search, Large Neighborhood Search
Learning to delegate [113]	CVRP	Regression	/	Improvement	Transformer	500, 1000, 2000	2000	LKH3, HGS-CVRP
Att-GCRN+MCTS [58]	TSP	Non-Autoregressive	/	Construction	Graph convolutional residual network with attention (Att-GCRN)	20, 50	990K	MCTS
NCO [20]	TSP	/	REINFORCE with critic baseline	Construction	LSTM encoder and LSTM decoder	20, 50, 100	/	Greedy Search, Stochastic Sampling, Active Search

Table 1.4: Summary of state-of-the-art NCO methods for routing problems (1/4).

Approach	Problems	Supervised learning	RL method	Type	Model	Instances size	Training dataset size	Search strategy
NCO for CVRP [132]	CVRP, SDVRP, SVRP	/	REINFORCE with critic baseline	Construction	1D CNN encoder and LSTM decoder	10, 20, 50, 100	33M	Greedy Search, Beam Search
NCO for HCVRP [180]	HCVRP	/	A2C	Construction	1D CNN encoder and RNN decoder	10, 20, 50, 80	n/a	Greedy Search
AM [100]	TSP, CVRP, SDVRP, OP, PCTSP, SPCTSP	/	REINFORCE with Greedy Rollout baseline	Construction	Transformer encoder and MHA attention decoder	20, 50, 100	128M	Greedy Search, Stochastic Sampling
AM-D [144]	CVRP	/	REINFORCE with Greedy Rollout baseline	Construction	Transformer encoder (dynamic embeddings) and MHA attention decoder	20, 50, 100	38M	Greedy Search, 2-OPT
SW-Ptr-Net, SW-AM [196]	TSP, CVRP	/	REINFORCE with Greedy Rollout baseline	Construction	Transformer encoder (dynamic embeddings) and MHA attention decoder	20, 50, 100	128M	Greedy Search
MRAM [198]	TSP, CVRP, SDVRP, OP, PCTSP, SPCTSP	/	REINFORCE with Greedy Rollout baseline	Construction	Transformer encoder and MHA attention decoder	20, 50, 100	128M	Greedy Search
Hybrid DRL & LS [206]	CVRP, CVRPTW	/	REINFORCE with Greedy Rollout baseline	Construction	MLP encoder and LSTM decoder	20, 50, 100	104M	Greedy Search, Beam Search, GLS, LNS
DRL-OPTW [60]	OPTW	/	REINFORCE with batch average baseline	Construction	Transformer with recursion encoder and LSTM decoder	20, 50, 100	500k	Greedy Search, Beam Search, Active Search
DRL-EVRPTW [114]	EVRPTW	/	REINFORCE with Greedy Rollout baseline	Construction	STRUCT2VEC encoder and LSTM decoder	5, 10, 20, 30, 40, 50, 100	1.28M	Greedy Search, Stochastic Sampling, Beam Search

Table 1.5: Summary of state-of-the-art NCO methods for routing problems (2/4).

Approach	Problems	Supervised learning	RL method	Type	Model	Instances size	Training dataset size	Search strategy
POMO [103]	TSP, CVRP	/	POMO	Construction	Transformer encoder and MHA attention decoder	20, 50, 100	20M	Multi-greedy Search
MDAM [197]	TSP, CVRP, SDVRP, OP, PCTSP, SPCTSP	/	REINFORCE with Greedy Rollout baseline and KL divergence	Construction	Transformer encoder (dynamic embeddings) and MHA attention decoders	20, 50, 100	128M	Greedy Search, Beam Search
JAMPR [51]	CVRP, CVRP with (soft, partly-soft, hard) Time Windows	/	REINFORCE with Greedy Rollout baseline	Construction	Transformer encoder (with tour and vehicle encoding) and MHA attention decoder	20, 50	51.2M	Greedy Search, Stochastic Sampling
E-GAT [111]	TSP, CVRP	/	REINFORCE with Greedy Rollout baseline	Construction	Edge-GAT encoder and MHA attention decoders	20, 50, 100	81.9M	Greedy Search, Stochastic sampling
GCN-NPEC [46]	CVRP	non-autoregressive	REINFORCE with Greedy Rollout baseline and binary cross-entropy	Construction	GCN encoder, GRU decoder and MLP decoder	20, 50, 100, 200, 400	50K	Greedy Search, Beam Search
S2V-DQN [93]	TSP	/	Deep Q-learning	Construction	STRUC2VEC	up to 300	n/a	Greedy Search
SOLO [135]	static and dynamic CVRP	/	Deep Q-learning	Construction	GNN	20, 50, 100	n/a	MCTS
OmegaZero [186]	TSP, CVRP	/	Deep Q-learning	Construction	GAT encoder and GRU decoder	500, 750, 1000 (TSP) 100, 200, 400 (CVRP)	n/a	MCTS
RP-DQN [16]	CVRP, MDVRP	/	Deep Q-learning	Construction	Transformer encoder (dynamic embeddings) and MHA attention decoder	20, 50, 100	n/a	Greedy Search, Stochastic Sampling

Table 1.6: Summary of state-of-the-art NCO methods for routing problems (3/4).

Approach	Problems	Supervised learning	RL method	Type	Model	Instances size	Training dataset size	Search strategy
Ztopping AM [12]	TSP, CVRP, SDVRP, OP, PCTSP, SPCTSP	/	REINFORCE with Greedy Rollout baseline	Construction	Transformer encoder and MHA attention decoder	20, 50, 100	128M	Ensembling with stochastic sampling
NLNS [80]	CVRP, SDVRP	/	REINFORCE with Baseline	Improvement	MLP encoder and MLP decoder	20, 50, 100	64M	LNS, LNS with Annealing acceptance criterion
NeuRewriter [30]	CVRP	/	Q-Actor-Critic	Improvement	Bi-directional LSTM encoder and MLP decoder	20, 50, 100	n/a	Greedy Search
NeuLNS [61]	CVRP, CVRPTW	/	Actor-Critic	Improvement	EGATE encoder and GRU decoder	100	25.6M	LNS
Learning Improvement Heuristics [195]	TSP, CVRP	/	n-step Actor-Critic	Improvement	Transformer with MLP	20, 50, 100	768K	2-OPT, SWAP
DPR [29]	CVRPTW	/	Proximal Policy Optimization	Improvement	HRGCN encoder and GRU decoder	25, 50, 100, 200, 400, 600, 800	n/a	LNS
DACT [121]	TSP, CVRP	/	Proximal Policy Optimization	Improvement	Transformer	20, 50, 100	n/a	Multi-greedy Search, 2-OPT, SWAP, RELOCATE
LCP [95]	TSP, CVRP, PCTSP	/	REINFORCE with Greedy Rollout baseline	Improvement	Transformer encoder and MHA attention decoder	20, 50, 100	128M	Stochastic Sampling

Table 1.7: Summary of state-of-the-art NCO methods for routing problems (4/4).

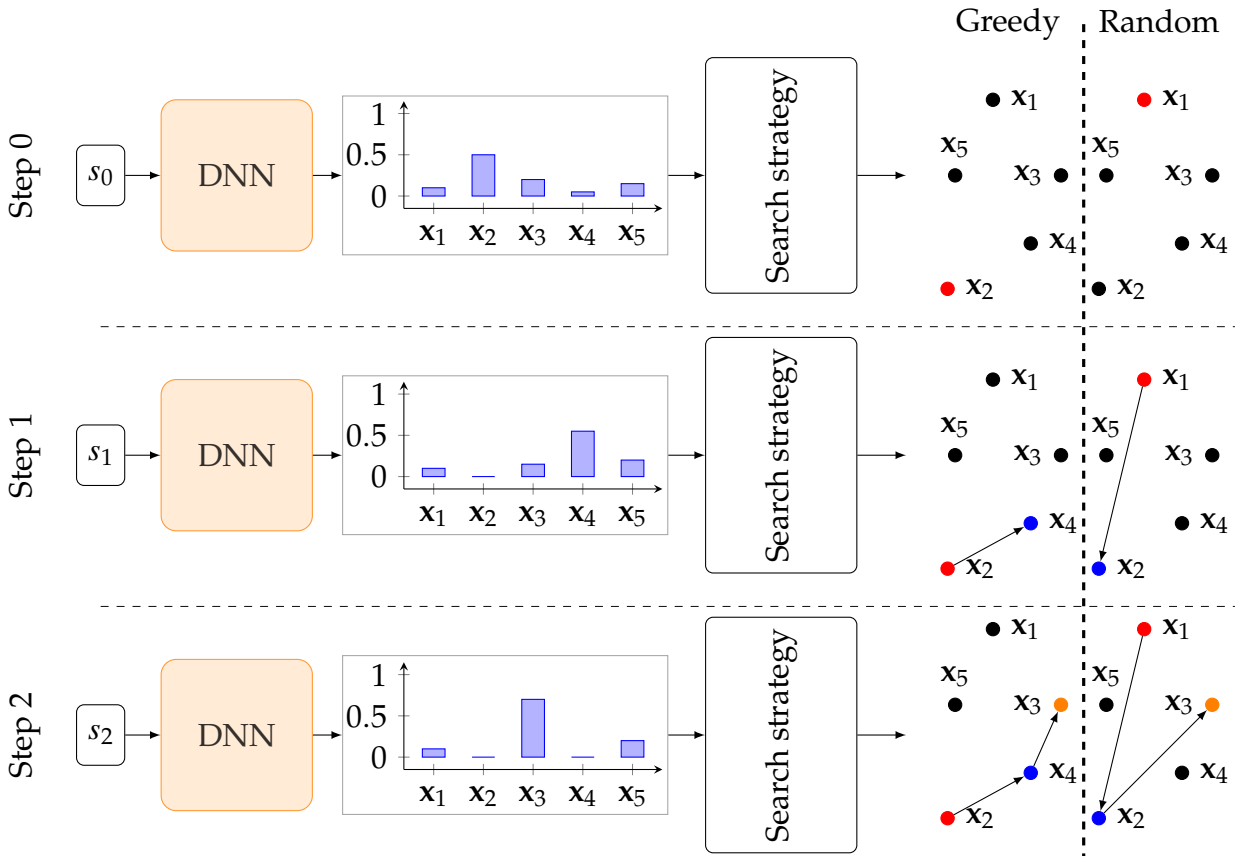


Figure 1.15: Example showing the first 3 steps of using greedy and random search strategies in the NCO framework on a 5 cities TSP instance.

search strategy, $\mathbf{x}_2$ is chosen because it has the highest probability of being selected. The random strategy chooses a city randomly based on a uniform distribution; $\mathbf{x}_1$ is selected. At step 2, s_1 contains information about the cities and the partial tour under construction which contains the city $\mathbf{x}_2$. The city $\mathbf{x}_4$ is chosen to extend the partial tour using the greedy search because it has the highest probability, while $\mathbf{x}_2$ has a zero probability because it was already included in the partial solution. The random strategy chooses the $\mathbf{x}_2$. Following the same procedure, at step 3, $\mathbf{x}_3$ is chosen by the greedy strategy while the random search happens to choose the same city.

The greedy strategy has the advantage of being fast to execute, but it shows

limited abilities to render the best solutions. To overcome this, [103] proposed a multi-greedy inference strategy. Instead of using a unique representation of an instance, the deep neural network is fed with numerous instances equivalent to the test instance. These equivalent instances are generated by performing isometric transformation on the coordinates (rotation, flipping, etc.), similar to data augmentation strategies developed in the computer vision field. Thus, the model has multiple views of the same instance, which may lead to alternative solutions. Finally, the best among decoded solutions is selected as the incumbent. This strategy was also used in DACT [121], which shows that the approach is valuable for both constructive and improvement methods.
Greedy Search has been used in most of the works done on Neural Combinatorial Optimization approaches, for its simplicity and its guarantee to output a valid solution. Among the surveyed works, 21 of them used this search strategy (see Tables 1.4–1.7).

Stochastic sampling. Introduced by Bello et al. [20], this strategy samples a node according to the probability distribution given by the deep neural network at each step, as it is done during training.

$$y_t \sim p_\theta(\cdot|y_1, \ldots, y_{t-1}, X), \quad \forall t \in \{1, \ldots, T\}$$

Using this search method, many candidate solutions can be sampled for a single instance. Thus, this increases the chances of finding a better solution. The diversity of the candidate solutions can be controlled by changing the clipping value C of the attention scores, to increase or decrease the entropy of the probability distribution [20]. Let us note that this strategy is also fast, Kool et al. [100] reports the generation of 1280 candidate solutions in less than one second for the Attention Model for TSP instances with 20 cities.

Beam search. This is a limited breadth first search greedy algorithm, which is controlled by a parameter ω named *beam width*. From a global view, this search algorithm keeps track of the most probable candidate solutions. At step 1, ω nodes with the highest conditional probability are selected. Each of these nodes will

start one of the ω candidate solutions. Starting from step 2, the joint probability between the selected nodes of each of the retained partial candidate solutions, and remaining non-selected nodes will be computed. The highest ω joint probabilities are selected. The process continues until all nodes are routed. The Greedy search can be considered as a special case of Beam search, where the beam width is equal to 1.

In Ptr-Net, the authors suggested using beam search to prevent from selecting invalid tours, for the TSP (e.g., tours with a city being repeated twice) [185]. Nazari et al. [132] obtained their best results using this strategy, with a beam width $\omega = 10$, while slightly increasing the computation time. In the MDAM approach, the beam search strategy is used on each of the decoders of the model. Thus, allowing for the exploration of more candidate solutions, e.g., a beam width $\omega = 10$ and a 5 decoders, beam search produces 50 candidate solutions.

Deep Policy Dynamic Programming (DPDP) is a novel search algorithm which is based on beam search [99]. The authors describe it as *"a beam search over the DP state space"*. The algorithm maintains a beam of candidate solutions and at each step, they are (1) expended by choosing one node, (2) removed from the beam if dominated, (3) updated with the best ω partial solutions. The particularity of this method is the addition of the concept of dominated solution. For the CVRP, a partial solution A dominates another partial solution B if both solutions visit the same nodes, while A has better cost, in terms of objective function, and better remaining vehicle capacity than B. In other words, they visit the same nodes, but differ in the order in which they do the visits. This strategy generates solutions which have a gap of 1.7% from the solutions found by HGS-CVRP [181] while being 6 times faster. Increasing the beam size makes this gap falls to 0.4% with a worse computation time (around 48 hours to solve 10000 test instances vs. 6 hours for HGS-CVRP).

Monte Carlo Tree Search. This tree search algorithm is one of the core algorithms in modern deep reinforcement learning to achieve super-human performances on difficult combinatorial games such as go, chess, and shogi [161]. The algorithm was used as a search method for Neural Combinatorial Optimization models. Fu et al. [58] tackled large TSP instances using a graph sampling al-

gorithm combined with a non-autoregressive model to generate heatmaps. The purpose of the graph sampling algorithm is to feed the large instance into a small-scale model (e.g., model trained to solve TSPs with 20 cities) piece by piece. The resulting heatmaps are merged to obtain the large instance's heatmap. This latter is exploited inside a MCTS algorithm to decode the final solution. OmegaZero [186] uses the MCTS algorithm similar to AlphaZero [161] to tackle TSP and the CVRP, to tackle instances up to 1000 cities (TSP) and 400 clients (CVRP).

Ensemble methods. In [12], the authors have observed that several models trained on the same dataset encodes different heuristics. While they have similar performance in the validation set, the subsets in which they perform well vary significantly. So, they proposed zero training overhead portfolio (Ztop), a method that trains several models instead of only one, as in ensemble learning, and selects the best k models that achieve the same performances on the validation set. At inference, the selected models are run on each test instance, and the best output, in terms of total travel distance, is selected. In terms of training, the method follows the same protocols as in single model training. The method was used with a set of trained Attention Models. The results showed a significant improvement over the use of a single Attention Model, while not adding any complexity during the training phrase.

Active search. This search algorithm (see Algorithm 4) is an improvement of the stochastic sampling strategy [20]. Rather than ignoring the rewards when sampling from a model, they can be exploited by the model to update the neural network weights towards a better optimum. The particularity of this algorithm is that it can start from a neural network with random weights, then trains it on a single instance. Thus, the model can process instances that are not from the same distribution. However, in all the cases, the algorithm overfits the instance being processed, so the resulting weights after the completion of the algorithm, generally can't be used on other instances. POMO [103] can be viewed as an extension of this approach for training on different instances. Instead of processing one instance at a time, a batch of instances is processed by generating a set of candidate solutions, much like in Active Search. However, contrary to Active Search, POMO does not

overfit and the solutions it can tackle are distribution-dependent. Let us also note that POMO is a training algorithm, while Active Search is a search algorithm intended to find the best solution for a single instance. Finally, Active Search opts for an exponential moving average (see lines 16-21 in Algorithm 4) rather than POMO's mean reward as a baseline. This strategy turned out to be more effective than stochastic sampling for the TSP [20]. It was also used as a search method for the OPTW [60] and the CVRP [79]. However, let us note that Active Search takes more time to execute than the other search methods.

Algorithm 4: Active Search

1 **Inputs:** Instance X, policy network P_θ, number of candidates K, batch size B, parameter β
2 $Y^* \leftarrow$ RandomSolution()
3 $C^* \leftarrow L(Y^*, X)$
4 $T \leftarrow \lceil \frac{K}{B} \rceil$
5 $Y[B]$ `// Empty array of size B`
6 **for** $t \leftarrow 1$ *to* T **do**
7 **for** $i \leftarrow 1$ *to* B **do**
8 $Y[i] \leftarrow \text{SampleSolution}(P_\theta, X)$
9 **end**
10 $j \leftarrow \arg\min_{k \in [1,B]} L(Y[k], X)$
11 $C \leftarrow L(Y[j], X)$
12 **if** $C < C^*$ **then**
13 $Y^* \leftarrow Y[j]$
14 $C^* \leftarrow C$
15 **end**
16 **if** $t = 1$ **then**
17 $b \leftarrow \frac{1}{B} \sum_{i=1}^{B} L(Y[i], X)$
18 **end**
19 **else**
20 $b \leftarrow \beta \cdot b + (1 - \beta) \cdot \frac{1}{B} \sum_{i=1}^{B} L(Y[i], X)$
21 **end**
22 $\nabla_\theta J_\theta \leftarrow \frac{1}{B} \sum_{i=1}^{B} (L(Y[i], X) - b) \nabla_\theta \log P_\theta(Y[i]|X)$
23 $\theta \leftarrow \text{Adam}(\theta, \nabla_\theta J_\theta)$ `// update the weights`
24
25 **end**

1.4 Evaluation protocols and results on CVRPLib

1.4.1 Current evaluation protocols

The evaluation protocol of neural combinatorial optimization for routing problems differs from the one that is used to evaluate handcrafted methods. Firstly, deep

neural networks require training, thus a sufficient number of training data must be provided. This number is beyond the number of available instances in any VRP benchmark set. Tables 1.4–1.7 give an overview of the number of data used to train deep neural networks for VRPs. Most of them require millions of data. Thus, most of the approaches are evaluated by generating synthetic VRP instances. This data generation protocol proposes to sample from a uniform distribution the necessary data. For example, for the CVRP, Nazari et al. [132] proposed to generate the clients and depot coordinates using a uniform distribution on $[0, 1] \times [0, 1]$, i.e, $\mathcal{U}\Big([0, 1]^2\Big)$, while the client's demands are integers between 0 and 9 chosen randomly following a uniform distribution. The vehicle's capacity Q is fixed according to the size of the instances tackled, $Q = 30, 40, 50$ for respectively CVRP with 20, 50 and 100 clients [132]. To date, it is the most widely accepted evaluation approach for the CVRP and is used in [103, 100, 144]. Secondly, although most models are agnostic to the size of the instances (i.e., we do not need to indicate the size of the instances that are tackled), they are usually trained on instances of fixed size, resulting in specialized models with limited generalization abilities, i.e., a model trained on instances of size 20 will perform well on other instances of the same size, but there is no guarantee that it will be good on instances of different sizes [88]. The reason they are trained with instances of a fixed size is mainly for convenience, as batches of the same size are required when training deep neural networks. Furthermore, regarding instance sizes, little work considered instances with sizes over 100 for the CVRP. This is due mainly to memory limits on GPUs used. In addition, the time required to train a model grows considerably as the size of instances becomes more important. Moreover, too little work considered real-world instances [46]. Finally, other works manage to find existing instance generators, such as the DIMACS TSP Challenge instance generator [86], which was used in the work of Khalil et al. [94].

1.4.2 Evaluation on CVRPLib instances

To close this chapter, we propose to compare the performance of NCO to well-established heuristics on CVRPLib instance sets [175][16]. All the gaps are measured

[16] available at: http://vrp.galgos.inf.puc-rio.br/index.php/en/

against the best solutions achieved by HGS [181], as a state-of-the-art metaheuristic for the CVRP, which gives the best known solutions for the sets (A, B, E, F, M, P, X).

CVRPLib gathers a collection of instance sets of the CVRP. These instances have different characteristics: uniformly distributed clients, clustered clients, small, medium and big size instances, etc. Thus, CVRPLib contains instances with heterogeneous properties that make them suitable for evaluating the performance of NCO models. This diversity of instances allows us to estimate more accurately the performances of the deep neural networks compared with instances that possess homogeneous characteristics (see Table 1.8). For example, the instances of the set A, E and P are small size instances with clients that are uniformly distributed. Thus, evaluating on them would confirm to us the performances of neural combinatorial approaches observed on the synthetic datasets. On the other hand, evaluating on the sets B and F would provide us information on the behavior of the models on small size clustered instances. Finally, evaluating on M and X[17] would give us an estimation of their performance on medium to big size instances.

set	clients' distribution	instances sizes	number of instances
A	uniform	32 to 80	27
B	clustered	31 to 78	23
E	uniform	22 to 101	11
F	clustered with clients close to each other	45, 71, 135	3
M	2 clustered (101, 121), 2 uniform (151, 200)	101, 121, 151, 200	4
P	uniform	16 to 101	24
X	mix of uniform and clustered	100 to 1000	1000

Table 1.8: Summary of CVRPLib instances.

To evaluate the performance of Neural Combinatorial Optimization framework with CVRPLib instances, we have reproduced the Attention Model [100], which we call NCO-AM for distinction. We used the Active Search algorithm, a single instance search method, for a better exploration of the search space for each instance. We compared the results with handcrafted heuristics: Route-first cluster-second, Nearest Neighbor and Sweep. We summarize our primary results in Table 1.9. The

[17] In our tests, we only used instances of size from 100 to 242 (31 instances) due to computational limits.

detailed results for each instance are available in Appendix A. Table 1.11 offers an overview of the results on a sample of three instances per set and their associated solutions, aiming to exhibit the distinctive performance traits of each instance set.

As we can see, NCO-AM consistently demonstrates the best performance on most sets, achieving the lowest optimality gaps on sets A, B, E, F, P, and M. However, it performs significantly worse in the set X with an average optimality gap of 90.09%, but slightly better than the Sweep algorithm. The average optimality gaps on the sets A, E and P confirm the performance observed in the literature by using random instances with clients uniformly distributed across the plane. Indeed, the results observed for small size instances are of the same order of magnitude as the ones we find on CVRPLib. Regarding the sets B and F, the model exhibits good performance on clustered instances, confirming that it can handle clustered instances. However, let us note that instances of set F are harder to solve since clients in the same cluster are really close to each other, increasing the risk of getting trapped in local optima. Besides this, the order of magnitude of the clients' demands varies significantly in the set F. For example, for the instance F-n72-k4, the minimum demand is 1 while the maximum is 21611. This strong variation may cause the learning process of a deep neural network to converge towards a worse local optimum. Regarding the size of instances, we can also see that the model performs well on small size instances (sets A, B, E, P) with an average optimality gap less than 6%. The performance decreases for medium to big size instances of the sets M and X. In the set X, there are instances where the model performs well, e.g., X-n148-k46 with an optimality gap of 0.91%, while other instances such as X-n101-k25 are harder to tackle. We make the same observation as for the instances of the set F, as it appears that instances with a strong variation in clients' demands are harder to solve.

Considering execution times (see Table 1.10), they are significantly longer than those of the best heuristic and metaheuristic approaches. Whether it is RFCS, Nearest neighbor, or Sweep, they only take a few seconds to output a solution, while for NCO-AM we measure its execution time in minutes. However, it should be noted that our study focused specifically on intensifying the search space exploration via the Active Search algorithm. Therefore, we used these models differently from their primary use, which is to be trained on a large database of instances and

Set	NCO-AM (±std)	RFCS	Nearest Neighbor	Sweep
A	**6.18** (±6.12%)	14.52	23.82	47.20
B	**4.42** (±1.97%)	10.50	20.43	23.10
E	**3.09** (±2.83%)	14.70	24.28	48.58
F	**10.16** (±4.23%)	11.78	27.26	71.92
P	**2.77** (±2.09%)	14.99	22.50	42.35
M	**7.38** (±3.37%)	16.76	29.44	108.14
X	90.09 (±126.93%)	**15.34**	18.99	108.74

Table 1.9: Comparison between average gap with HGS solutions per set of CVRPLib instances (%).

Set	Avg. execution time (min.) (± std (min.))	Avg. time to best solution (min.) (± std (min.))
A	126.72 (±32.64)	81.01 (±49.80)
B	130.62 (±31.04)	103.85 (±46.54)
E	185.19 (±126.39)	151.34 (±135.55)
F	273.09 (±226.05)	102.11 (±61.41)
P	131.72 (±73.94)	71.63 (±55.86)
M	595.16 (±206.23)	377.46 (±272.99)
X	722.27 (±188.22)	549.79 (±261.93)

Table 1.10: Summary of average total execution times and average time until the best solution for NCO-AM on CVRPLib sets.

then used for inference on unseen instances during training, with exploitation algorithms that can be executed in a few seconds or minutes. As expected, the model takes longer to run as the size of the instances increases. Furthermore, it is noteworthy that the model consistently identifies its best solutions considerably earlier than the completion of the entire execution process. This observation implies that the model converges towards a local optimum well in advance of the total execution time.

Instance	Q	K	Tightness	q_{min}	q_{max}	BKS	obj.	gap (%)	t_{tot}	t_{best}
A-n32-k5	100	5	0.82	1	24	784.0	787	0.38	83.56	19.83
A-n53-k7	100	7	0.95	1	30	1010.0	1064	5.35	132.33	47.06
A-n80-k10	100	10	0.94	1	26	1763.0	1812	2.78	204.86	196.44
B-n35-k5	100	5	0.87	1	26	955.0	989	3.56	91.36	80.68
B-n50-k7	100	7	0.87	2	63	741.0	769	3.78	127.16	123.67
B-n78-k10	100	10	0.94	1	26	1221.0	1302	6.63	201.93	199.88
E-n22-k4	6000	4	0.94	100	2500	375.0	375	0.00	63.04	2.96
E-n76-k10	140	10	0.97	1	37	830.0	847	2.05	202.38	192.71
E-n101-k14	112	14	0.93	1	41	1067.0	1182	10.78	423.10	417.99
F-n45-k4	2010	4	0.9	1	1300	724.0	798	10.22	113.27	56.53
F-n72-k4	30000	4	0.96	4	21611	237.0	251	5.91	174.28	171.95
F-n135-k7	2210	7	0.95	1	1126	1162.0	1329	14.37	531.73	77.84
M-n101-k10	200	10	0.91	10	50	820.0	852	3.90	413.03	105.53
M-n151-k12	200	12	0.93	1	41	1015.0	1088	7.19	604.60	603.34
M-n200-k17	200	17	0.94	1	41	1282.0	1365	6.47	880.79	621.35
P-n16-k8	35	8	0.88	6	31	450.0	450	0.00	54.59	1.69
P-n55-k10	115	10	0.91	5	37	694.0	709	2.16	145.23	119.72
P-n101-k4	400	4	0.91	1	41	681.0	727	6.75	399.17	40.09
X-n101-k25	206	25	1.0	1	100	27591.0	51464	86.52	469.46	445.97
X-n148-k46	18	46	0.99	1	10	43448.0	43844	0.91	710.73	670.28
X-n242-k48	28	48	0.99	1	10	82751.0	86007	3.93	1086.66	1065.88

Table 1.11: Sample of CVRPLib instances and their corresponding solution found by NCO-AM. Q: vehicle's capacity, K: number of routes, Tightness = $\frac{\sum_{i=1}^{n} q_i}{KQ}$; $q_{\min}$: the minimum demand; $q_{\max}$: the maximum demand; BKS: HGS solution; $obj.$: solution's objective value found by AM-NCO; gap: percentage of gap to the BKS ($(1 - \frac{obj.}{BKS}) \times 100$); t_{best}: cpu time to find $obj.$ in minutes; t_{best}: total runtime in minutes.

1.5 Conclusion

In this chapter, we have laid down the essential foundations that will serve as a basis for developing the rest of this thesis. First, we introduced the Capacitated Vehicle Routing Problem (CVRP) and its variants. We also discussed various handcrafted solution methods commonly used to solve CVRP, including exact algorithms, heuristics and metaheuristics.

Next, we introduced deep learning, as a subfield of machine learning that uses artificial neural networks to model complex nonlinear functions. We presented various types of neural networks used in this work, including recurrent neural networks (RNNs), transformers, and graph neural networks (GNNs). These networks are known for their ability to capture complex dependencies in sequential data or graph structured data. Moreover, we outlined reinforcement learning, as another subfield of machine learning in which an agent interacts with its environment and learns from the feedback it receives.

We reviewed the use of deep learning and reinforcement learning to solve VRPs. This involved discussing various approaches that have been used in the literature, with deep reinforcement learning emerging as today's dominant training paradigm. We also proposed a distinction between construction methods and improvement methods, and we classified state-of-the-art contributions according to this distinction. We highlighted that these models need an important amount of data to be efficiently trained.

Finally, we presented the primary results of our experiments with the CVRPLib instances by reproducing the Attention Model [100]. Deep learning models show promising results, suggesting that it may be able to solve routing problems efficiently without specifying any prior knowledge on how to achieve this. However, execution times are still a challenge, and there is still room for improvement for medium and big size instances. In the following chapters, we present our contributions in the light of our main findings in this chapter. Regarding the amount of data, we propose to study in Chapter 2 transfer learning to exploit prior knowledge to reduce the number of training example needed to tackle the CVRP. Chapter 3 presents a two-steps method for tackling the CVRP instead of the single-step construction methods presented in this chapter. Finally, Chapter 4 is dedicated to

a dynamic and stochastic problem to further explore the NCO framework.

Chapter 2

Transfer learning under the Neural Combinatorial Optimization framework

One of the main lines of research in reinforcement learning is whether a policy learned to handle one task can be used to handle another different, but related task. In the field of machine learning, this problem is known as transfer learning. This is an interesting direction of research to reduce computation time and resource usage when it is expensive to train a new policy from scratch. In this chapter, we study transfer learning in the context of neural combinatorial optimization for routing problems. We first review the literature on transfer learning and the main metrics used to monitor the learning phase. We then describe how to use transfer learning in VRPs. We experiment with transfer learning from TSP to CVRP under different scenarios, and we compare the resulting policy with a policy trained from scratch. The study of this approach allowed us to identify cases where transfer learning is useful for learning neural combinatorial optimization policies.

2.1 Transfer learning

2.1.1 Transfer learning in deep learning

Transfer learning (TL) has its roots in the field of psychology, with the pioneering work of Woodworth and Thorndike [193]. They argue that when the stimulus-response elements share common properties between a source task and a target task, it is possible to use the knowledge acquired on the source task to learn the target one. The Machine Learning field takes inspiration from this concept and adapts it for learning models. In Machine Learning, transfer learning consists of the development of methods that transfer knowledge to a model from a source task (or domain) to a target task. This is used to introduce a bias into the learning process for the target task. This bias is derived from the pre-existing knowledge acquired by the model through training on the source task. The transfer of this *a priori* knowledge influences the learning process and guides the model's behavior when adapting to the target task. To guarantee the success of transfer learning, the source and the target tasks have to be *similar*. This similarity condition can be explained via the representations learned by the model on the source task, which are suited for the target task. Goodfellow et al. best describe this in Chapter 15 of the Deep learning book [65]: *"In general, transfer learning (...) can be*

achieved via representation learning when there exist features that are useful for the different settings or tasks, corresponding to underlying factors that appear in more than one setting." Thus, instead of starting the learning process from a random representation of the features of the target task, a representation resulting from a learning process on the source task is used.

Concretely, implementing transfer learning consists in reusing the values of the weights θ_S^* of a deep neural network (DNN) as initial values for the training of a target task $\mathcal{T}_T$ on a target dataset $\mathcal{D}_T$ (Equation 2.2). This weight refinement on the target domain is also known as *fine-tuning*. The weights θ_S^* result from the training on a source task $\mathcal{T}_S$ on the source dataset $\mathcal{D}_S$ (Equation 2.1).

When learning the source task, the neural network weights are randomly initialized.

$$(\mathcal{T}_S, \mathcal{D}_S, \theta_S) \overset{learning}{\longmapsto} \theta_S^* \tag{2.1}$$

$$(\mathcal{T}_T, \mathcal{D}_T, \{\theta_S^*, \theta_T\}) \overset{learning}{\longmapsto} \theta_T^* \tag{2.2}$$

Figure 2.1 illustrates the two previous equations and schematizes transfer learning. We can notice that, generally, only the values of the weights of the layers responsible for learning the representations are reused for learning to solve the target task. The output layer responsible for predictions (in red, in the figure, e.g. classification head), often specialized in the source task, is omitted and an output layer with random weights is used for that part in the learning of the target task. This is depicted in Equation 2.2 with additional weights θ_T. Note that the boundary between what constitutes the representation learning layers and the prediction layers can sometimes be difficult to determine. Empirically, this can be established by first copying a subset of the values of the weights learned by the source model into the target model and initializing the rest randomly. If an improvement is found, we iterate the process with additional weights initialized from the source neural network, otherwise we stop.

Transfer learning has been widely studied in the supervised learning case. It is interesting when it is difficult to get a large labeled dataset, either because labeling a large dataset is a tedious task, or not enough data samples are available.

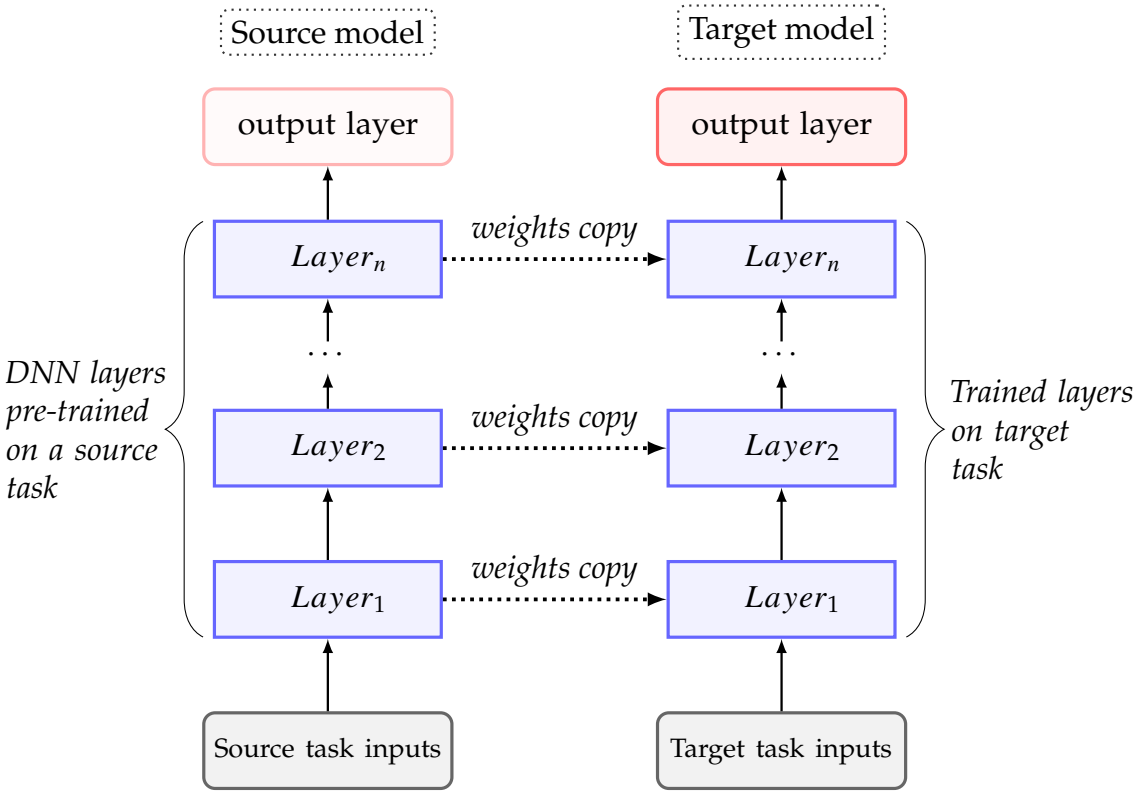


Figure 2.1: Overview of transfer learning.

In the case of deep neural networks, too little data is available to train a complex model from scratch. In this case, transfer learning can be used to train the model and improve the accuracy compared with training from scratch. There are no guarantees that transferring knowledge yields good results, due to *negative transfer* issues [187]. This may happen when the source and target domains have many dissimilarities, or the model trained on the source task may not be suitable for the target task. Therefore, the use of transfer learning calls for vigilance and a meticulous definition of the source and target domains as well as the model used.

In practice, transfer learning has been successfully applied for real-world glitch detection and exhibits that it reduces training time while improving the accuracy [63]. The power of transfer learning has also been demonstrated in numerous computer vision tasks ranging from image classification, scene recognition, to attribute detection and image retrieval [162]. In Natural Language Processing, it has been

demonstrated that learned features can be shared between multiple languages, and can improve the system performance on new languages [81]. Thus, these promising results on supervised learning encouraged the Machine Learning community to investigate more on transfer learning in deep reinforcement learning.

2.1.2 Transfer learning in reinforcement learning

As defined in the recent survey of Zhu et al. [210], transfer learning in reinforcement learning aims to learn an optimal policy for the target task using knowledge from one or more source tasks. In this case, source and target tasks are represented as Markov decision processes. Knowledge transfer can take various forms, such as the reuse of a set of trajectories learned on the source task, or directly reuse the policies learned from the source tasks (*policy reuse* [55]). Thus, unlike the supervised learning case, in reinforcement learning, transfer learning can be exploited in several ways to bias the policy on the target task with knowledge from the source domain.

Although in reinforcement learning it is possible to learn without any form of supervision, the learning process usually requires several data samples, especially policy-based methods that are known for their sample inefficiency. Thus, transfer learning received much attention to improve the sample efficiency of reinforcement learning algorithms [108]. Moreover, the potential of transfer learning does not end there. Figure 2.2 illustrates the expected key contributions of transfer learning for reinforcement learning algorithms in a reward maximization setting. The plot in blue represents a model that learns from scratch without transfer learning, while the orange plot represents a model that learns with transfer learning. Figure 2.2-(a) corresponds to a setting where transfer learning improves the jump-start performance of the model, i.e., the initial performance on the target task. In this case, the model performs better on the target task without additional adaptation of its weights (zero-shot transfer). Figures 2.2-(b), (c) show a more general case where transfer learning improves the model's sample efficiency with a distinction between equal jump-start performance (Figures 2.2-(b)) and worse jump-start performance (Figures 2.2-(c)). In the case (b), the model performs few-shot transfer, i.e., the model needs few additional learning iterations to achieve better performance with

transfer learning. The case (c) is more general: the jump-start performance of the model with transfer learning is worse than the performance of the model trained from scratch, and it needs more training epochs than the few-shot transfer to achieve a better performance. Finally, Figure 2.2-(d) depicts the case where the final performance is improved; not only does the model need fewer iterations to reach the performance of the model trained from scratch, but the model also achieves better asymptotic performance when trained with transfer learning.

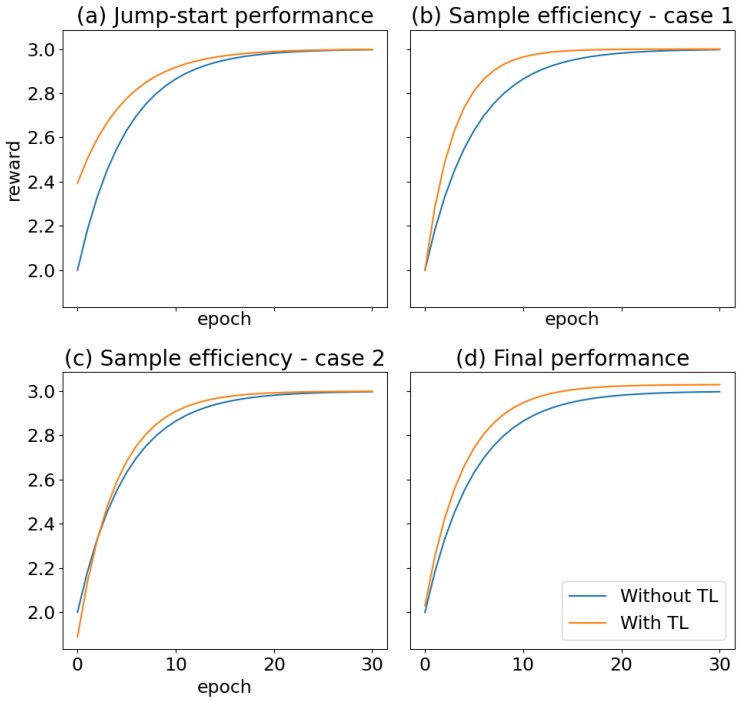


Figure 2.2: Contributions of transfer learning to reinforcement learning (higher is better).

Transfer learning has been successfully applied in reinforcement learning tasks. In robotics, for example, transfer learning is useful for pre-training an agent's policy on a simulator before being trained on real-world environments [201]. Training on a simulated environment gives the agent the advantage of doing more interactions with it. Moreover, it reduces costs, since only the fine-tuning is done in the real world. Another significant example of transfer learning is *AlphaGo* for board games, which was pre-trained on a database of expert demonstrations. Its policy

is then optimized via Monte Carlo Tree Search [165]. An important number of these applications involve pre-training by imitation learning, i.e., the reinforcement learning agent first learns from a set of expert demonstrations before adapting its policy to the downstream task.

Few attempts have been made to apply transfer learning to combinatorial optimization problems. Nevertheless, all the cases listed in the literature concern a special case of fine-tuning on the same problem, but with instances from a different distribution (e.g., a model trained on CVRP instances with random distribution of the clients may be fine-tuned to tackle other instances with clustered clients). For example, zero-shot transfer has been successfully applied in a scheduling task for different computation graphs [140]. Another interesting use case is reported in the study of transfer learning in genetic programming hyper-heuristic for the Uncertain Capacitated Arc Routing Problem [8]. The conclusions of the study indicate that transfer learning improves the jump-start performance by allowing the creation of a good new initial solution for the target problem. Furthermore, we report an application of transfer learning on memetic algorithms for routing problems, to solve new instances using knowledge learned from previously solved ones [54]. Finally, Active Search [20] can also be seen as a fine-tuning strategy which adapts a deep neural network's weights to a specific instance (regardless of its original distribution). For more detail on Active Search, readers may refer to the section 1.3.3 of this thesis.

To the best of our knowledge, none of the attempts in the literature involve transferring knowledge from one VRP problem to another more or less constrained one, as discussed in the following sections.

2.2 Motivations and goals

As highlighted in Tables 1.4–1.7 of Chapter 1, the majority of the reinforcement learning methods used in end-to-end learning involve policy-based methods. However, these methods are known for their sample inefficiency, which results in the need of millions of instances to derive a good quality policy. For example, the state-of-the-art approach of Kool et al. [100] uses 1.28M instances per training epoch. Therefore, in the current framework, whenever a new problem is formulated by

	TSP	CVRP
State	partial solution under construction and unvisited nodes	partial solution under construction and unvisited nodes
Action	choose an unvisited city	choose an unvisited client or the depot
Reward	total distance of the solution under construction (deterministic)	total distance of the solution under construction (deterministic)
Transition function	deterministic	deterministic
Termination condition	all cities are visited	all clients are visited

Table 2.1: Comparison between TSP and CVRP reinforcement learning MDP modeling.

adding or removing constraints, a new model is trained from scratch to solve it by policy-based deep reinforcement learning. Thus, the new problem's relationship to the previously solved problem is ignored. As well, from a practical side, there is no guarantee that enough data is available for the new problem to train a model from scratch. This could result in a model that converges to a poor local minimum. Hence, studying transfer learning in this case could provide interesting insights into the conditions of its applicability. Of course, we do not expect similar or better performance than a model trained from scratch with enough data and enough training epochs. We can then consider that we are interested in a *degraded mode*, where we do not have enough time, nor enough data to train on the target problem.

In this chapter, our case study concerns transfer learning in the Neural Combinatorial Optimization famework from TSP to CVRP, as two key representative routing problems. As seen in Chapter 1, the two problems are related, since the CVRP is a generalization of the TSP. Following the reinforcement learning formalism, the two problems differ only in the action space (see Table 2.1). For the TSP, an action corresponds to selecting a city not yet visited, while in the CVRP case, an action is either "selecting a client not visited", or "coming back to the depot". Thus, there exists an action in the CVRP that can be performed multiple times (visiting the depot), which makes the problem relatively more difficult to solve.

To carry out this study, we make few assumptions. First, we consider that the deep neural network model used on both TSP and CVRP are identical, i.e., they have the same number of parameters, and the same architecture is used on both

problems. Moreover, all weight values obtained during training on TSP are kept at the beginning of training on CVRP; no layer of the neural network has its weights randomly reset. These two assumptions put us in a situation where we consider a model well-trained for a problem (TSP) that must be adapted to a new problem (CVRP) with relatively less training data. Regarding the features, the CVRP uses the same raw features as inputs of the deep neural network. This assumption is implicitly included in the first one. Incorporating more features would necessitate the inclusion of extra neurons on layers within the neural network. As a result, additional parameters would be required to encode these features. Finally, we measure the performance of the model by training it on fewer epochs than the pre-training phase on TSP, with the aim of studying the adaptation of the model to the downstream task (CVRP). To test the relevance of transfer learning, we compare the model trained via transfer learning with the same model trained from scratch by monitoring the metrics introduced in the previous section; jump-start performance, asymptotic performance, performance with fixed training epochs.

Guided by our assumptions, we formulate research questions to which we will attempt to provide answers.

1. In connection with the first two assumptions, is the representation learned with TSP instances sufficient to learn a policy for the CVRP?

2. At what data count limit does transfer learning become useful?

3. How does the size of the dataset used during pre-training for the TSP influence the model's performance when it is reused for the CVRP?

4. Will transfer learning be effective if the TSP and CVRP instances are drawn from different distributions?

2.3 Model description

Our study is carried out on the NCO architecture proposed by Kool et al. [100] (see page 65) which has proven to be an adequate candidate for both TSP and CVRP. To fit our assumptions on the number of parameters and neural network

architecture, we made adaptations to the original model. In what follows, we briefly describe the model and introduce the adjustments we made.

This model is an encoder-decoder architecture that iteratively computes, at each step t, the probability of choosing a node x_i over the others from the problem's raw features. In the original implementation, the TSP raw features considered are the $2D$-coordinates of the cities, while for the CVRP, in addition to the clients coordinates there is also their respective demands. Furthermore, there is a distinction between the depot and the clients nodes. In our formulation, to avoid introducing additional parameters to the model, we use for both problems only the $2D$-coordinates as raw features for the encoder, while the demands are only used in the masking operation for the CVRP[1]. In the work of Nazari et al. [132], the authors mention that using the demands along with the embedding of the $2D$-coordinates for the decoder brings no significant improvement. Thus, we pushed further this idea by only using the clients' coordinates in the whole encoder-decoder approach. Figure 2.3 shows the overview of the Attention Model architecture with a linear embedding followed by an encoder and a decoder that we detail below.

2.3.1 The Encoder

The encoder is made of two main components: an embedding layer which maps the $2D$-coordinates to a vector in a $d-$dimensional space, and a transformer encoder [178] that acts like a graph neural network on instances (recall that here we consider an instance as a complete graph).

The embedding layer consists of a linear transformation of the node's $2D$-coordinates, following the equation: $\mathbf{h}_i = \mathbf{W_x}\ \mathbf{x}_i + \mathbf{b_x}\ \forall i \in \{0, \dots n-1\}$, with $\mathbf{h}_i, \mathbf{b_x} \in \mathbb{R}^d$, $\mathbf{x}_i \in \mathbb{R}^2$, $\mathbf{W_x} \in \mathbb{R}^{d\times 2}$, and n being the number of nodes in an instance. Unlike the original model, we do not use a separate linear embedding for the depot. Although it is a special node in the CVRP, using the same linear embedding used for the clients aims at benefiting from the representation of the coordinates learned while solving the TSP when performing transfer learning.

For the transformer encoder, it consists of N transformer blocks. The first

[1] in the original implementation [100], the demands are used along with the nodes coordinates to compute node embeddings.

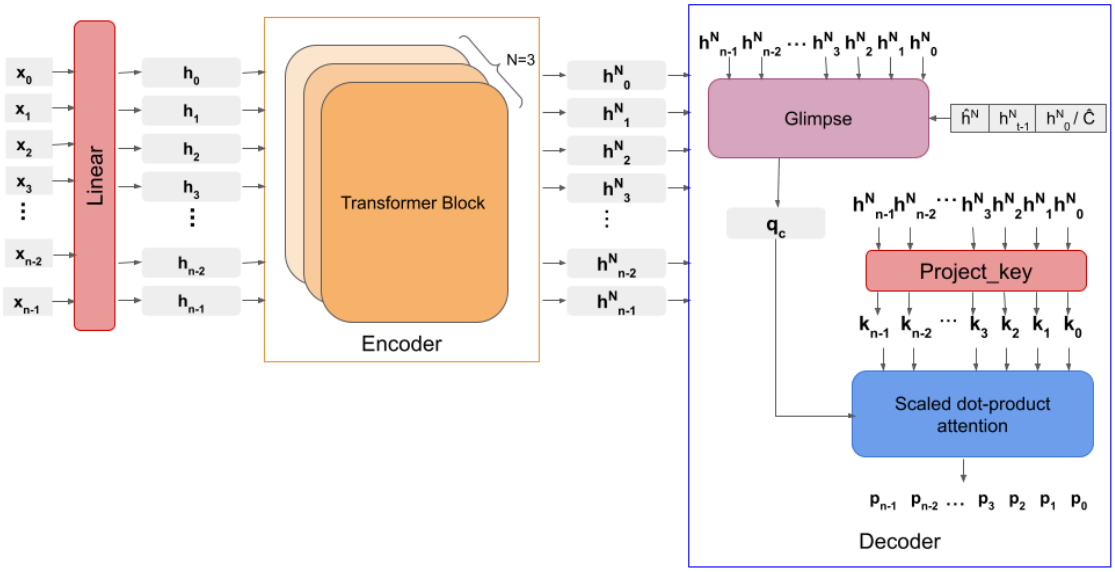


Figure 2.3: Overview of the Attention Model architecture used in our TL experiments.

block takes as input the resulting embeddings of the $2D$-coordinates, while the others take the output of the previous layer. Each transformer block consists of two components: a multi-head attention, a feed-forward layer and residual layers. We used 3 transformer blocks with 8 heads, as suggested in the literature [100]. The result of the encoder is a set of $d-$dimensional node embeddings; the final embedding of a node $\mathbf{x}_i$ is denoted $\mathbf{h}_i^N$.

2.3.2 The Decoder

The decoder is used to compute the probability distribution that measures the probability that a partial solution has to be extended by a given node in order to, in the end, minimize the total traveled distance. As we will explain hereafter, this probability distribution will be computed confronting a context embedding with the node embeddings. In the original model, the number of parameters defining the context embedding varies whether we are dealing with TSP or CVRP. Here, we also introduce modifications to preserve the same number of parameters for

both problems.

For the TSP, at each time step t, the context embedding $\mathbf{h}_c$ is defined as a $3d-$dimensional vector resulting from the concatenation of: (i) the representation of the graph ($\overline{\mathbf{h}}^N = \frac{1}{n}\sum_{i=0}^{n-1}\mathbf{h}_i^N$), (ii) the representation of the lastly selected node $\mathbf{h}_{y_{t-1}}^N$, and (iii) the representation of the first selected node $\mathbf{h}_{y_0}^N$. At $t = 0$, two parameter vectors $\mathbf{v}^l \in \mathbb{R}^d$ and $\mathbf{v}^f \in \mathbb{R}^d$ are used to substitute the undefined values of (ii) and (iii). The context embedding is thus defined as follows[2]:

$$\mathbf{h}_c = \begin{cases} [\overline{\mathbf{h}}^N; \mathbf{h}_{y_{t-1}}^N; \mathbf{h}_{y_0}^N] & t \geq 1 \\ [\overline{\mathbf{h}}^N; \mathbf{v}^l; \mathbf{v}^f] & t = 0 \end{cases}$$

For the CVRP, the context embedding $\mathbf{h}_c$ is the concatenation of the representation of the graph $\overline{\mathbf{h}}^N$, the representation of the lastly selected client (at $t = 0$, we use the representation of the depot $\mathbf{h}_0^N$) and the representation of the remaining capacity. In the original implementation, this results in a $(2d + 1)-$dimensional context vector $\mathbf{h}_c$. However, to keep the same $3d-$dimensional embedding for the CVRP as in the TSP, we did not use the remaining vehicle capacity $C \in \mathbb{R}$ as is as a feature. Instead, we compute an embedding of the remaining capacity by diverting the vector $\mathbf{v}^f$, i.e. $\hat{\mathbf{C}} = C\ \mathbf{v}^f$, thus at each time step t:

$$\mathbf{h}_c = [\overline{\mathbf{h}}^N; \mathbf{h}_{y_{t-1}}; \hat{C}]$$

.

A glimpse mechanism [20] (see page 63) is then used to enhance the context vector into a $d-$dimensional query vector $\mathbf{q}_{(c)}$ that will be used by an attention mechanism to compute the probability of selecting a node as the next node. This mechanism consists of a multi-head attention that computes the context vector interaction regarding the nodes that have not yet been selected. Finally, a scaled dot product attention mechanism [178] is used to compute the probability of choosing a node over another. This dot product uses the query vector $\mathbf{q}_{(c)}$ and the keys $\mathbf{k}_i = \mathbf{W}_K\ \mathbf{h}_i^N,\ \forall i \in \{0, \ldots, n-1\}$ that result from a linear projection of the node embeddings.

[2]We recall that $[\cdot;\cdot]$ is the concatenation operation.

A mask is used to prevent selecting already selected nodes, as described in Chapter 1. For the TSP, it is straightforward, as we mask already visited cities. For the CVRP, the masking policy is more complex. Computing the probability distribution defining the node to visit at t, we mask:

- Clients with demands greater than the remaining capacity;
- Already satisfied clients;
- The depot, if it has been selected at $t-1$, to prevent selecting it at the current iteration.

2.4 Experimental protocol

2.4.1 Model training

We trained the model using reinforcement learning, as it tends to express interesting generalization properties that may benefit Transfer Learning [90]. Supervised learning would also be an option for the TSP, e.g., training our model from expert demonstrations using the Concorde solver [6]. However, reinforcement learning does not require labelled data that may be time-consuming to obtain. The parametric policy is more particularly trained using REINFORCE with a greedy rollout baseline, as used in Kool et al. [100].

2.4.2 Model pre-training on TSP

We first pre-trained our model to solve the TSP. We used the same data generation and training protocol commonly used in the literature, for each training epoch, 2500 batches of 512 instances have been generated (1.28M instances per epoch). The data generation process consists of uniformly drawing $2D$-coordinates from the unit square, i.e., $\mathcal{U}\left([0,1]^2\right)$ unit square. For the validation, we used 10k instances sampled from the same uniform distribution. Two models named TSP20 and TSP50 have been trained considering different datasets composed of instances of 20 and 50 cities respectively. Both models have been trained during 100 epochs

using the Adam optimizer [96] with a learning rate of 10^{-4}. These models reach state-of-the-art performance reported in Kool et al. [100].

2.4.3 Transfer learning to CVRP

Guided by our hypothesis and research questions, we, hereafter, present our different experiments on transfer learning for the CVRP. Regarding the number of training data, we consider three different settings: 16k, 32k and 64k instances per epoch. Considering these settings, our models are trained on respectively 80, 40 and 20 times fewer data per epoch than the original models. We also consider training on only half the number of epochs used for the TSP, i.e., 50 epochs.

For each model trained with transfer learning, we consider its counterpart trained without transfer learning, for comparison. When trained without transfer learning, the models are trained from randomly initialized parameters using the Adam optimizer with a learning rate of 10^{-4}. When transfer learning is applied, the learning rate is set to 10^{-5} to exploit the optimizer's saved state from the TSP, and to gradually adapt the model's parameters to the CVRP.

We generate CVRP instances following the protocol of Nazari et al. [132] by sampling $2D-$coordinates from $\mathcal{U}\Big([0,1]^2\Big)$. The demands d_i of clients ($i \in \{1,\ldots,n-1\}$) are uniformly drawn from $\{1,\ldots,9\}$. Vehicle capacities are 30 for instances with 20 nodes and 40 for instances with 50 nodes.

Our experiments are based on several factors: the data distribution of the instances of the two problems, the instance size of the TSP and CVRP instances, as well as the number of data used in the pre-training and the transfer learning phases. We derive four series of experiments labelled as follows:

1. **Same**: the size of the instances and the data distributions are the same for the TSP and the CVRP; we use a uniform distribution to sample nodes coordinates for the TSP and CVRP instances. For example: we pre-train on TSP with 20 cities drawn from a uniform distribution, and we train on CVRP with 20 nodes drawn from a uniform distribution.

2. **Diff size**: the size of the instances differ between the TSP and CVRP, and the data distributions are the same. For example, we pre-train on TSP with

20 cities drawn from a uniform distribution, and we train on CVRP with 50 nodes drawn from a uniform distribution.

3. **Diff Dist**: the size of the TSP and CVRP instances is the same, but the data distribution is different. For example, pre-training is done on TSP with 20 cities drawn from a uniform distribution, and training is done on CVRP with 20 cities drawn from a normal distribution.

4. **Pretext**: The term pretext task is borrowed from the literature of self-supervised learning [122]. It is a task designed to create a proxy that enables the model to learn useful representations from the dataset of the downstream task. For example, it has been successfully used to pre-train large language models, such as GPT-3 [26]. In our case, we use the CVRP instances to first train a policy for the TSP. Then, starting from the resulting model and the same CVRP instances, we use transfer learning to learn a policy for the CVRP. The goal is to learn, from the pretext task (TSP), representations that will be useful for the resolution of the target task (CVRP). Our study aims, in this case, at knowing if the use of the TSP as a pretext task brings a gain for the creation of better models for the CVRP. For example, we pre-train the model to learn a policy for the TSP on 32k CVRP instances with 20 cities drawn from a uniform distribution, then, we train the model to learn a CVRP policy on the same 32k instances.

2.5 Results and discussion

2.5.1 Pre-training results

Two models, respectively denoted TSP20 and TSP50, are trained on instances of 20 and 50 cities during pre-training on TSP. Figure 2.4 depicts the evolution of average solution length per epoch on the validation set during the training phase in both cases. Validation is performed using 10k TSP instances unseen during training, with 20 and 50 cities respectively. We observe that the average solution length decreases, which implies that the model successfully learns to solve the TSP for both instance sizes, and converges towards a minimum. Besides this, our

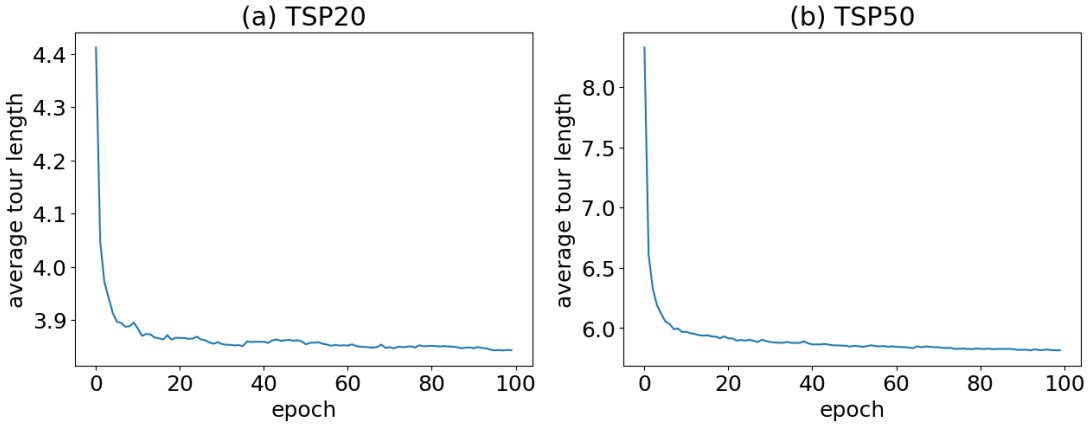


Figure 2.4: Evolution of the average tour length per epoch in validation of the model on TSP.

version of the model achieves average tour lengths of 3.84 and 5.82 for TSP 20 and 50 respectively, which matches the performance of the original model of 3.85 and 5.80 on the same instance sizes [100], despite introduced modifications.

2.5.2 Transfer learning results

We now present the results obtained in the different settings.

Same – Applying transfer learning using models pre-trained on TSP with the same size and data generation distribution as the one considered for the CVRP instances. Figure 2.5 shows the average tour length of CVRP models during the training phase. Both models are presented: (i) trained from scratch with a random weight initialization (NO-TL[3] in blue), and (ii) trained starting from a pre-trained TSP model (TL in orange). Plots a-b-c correspond to VRP20 with transfer learning from TSP20 models, while plots d-e-f correspond to VRP50 with transfer learning from TSP50 models. Each plot corresponds to a number of instances per epoch used to train our model, respectively from left to right, 16k, 32k and 64k.

Three phases can globally be distinguished in the learning process:

- Epoch 0 to 5: initial phase, the learning curve is rather in favor of models

[3]NO-TL designates the model trained from scratch: No-Transfer learning.

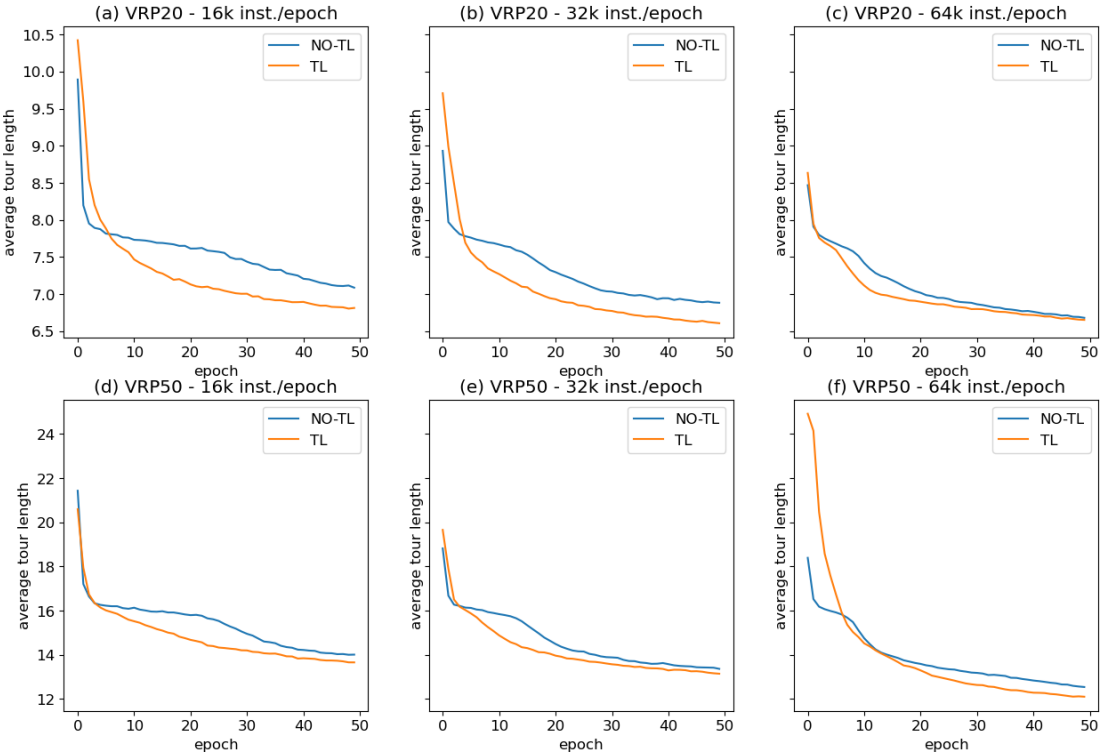


Figure 2.5: Comparison of the evolution of average tour lengths per epoch in training between CVRP models trained without Transfer Learning (NO-TL/blue) and CVRP models trained using transfer learning (TL/orange) with different number of instances per epoch (16k, 32k, 64k) with 20 nodes (plots a, b and c) and 50 nodes (plots d, e and f).

without transfer learning, with a rapid decrease in tour lengths for both types of models.

- Epoch 5 to 30: there is a shift in the training curves in favor of the transfer learning models. The average tour lengths per epoch decreases faster using transfer learning. The shift happens early in the learning process, so it appears that our models need few-shot samples to adapt to the CVRP domain.

- Epoch 30 to 50: the learning curves of the two models continue to decrease, but more slowly than in the previous phase. Transfer learning models still

outperform those trained without transfer learning.

Based on our findings, it appears that for the jump-start performance metric, our models do not perform zero-shot adaptation since the initial performance is poor compared with the NO-TL models. This can probably be partially explained by the fact that the context embedding $\mathbf{h}_c$ differs for the TSP and the CVRP, since the vector $\mathbf{v}^f$ was departed from its initial representation to encode the remaining capacity. The early shift seems to indicate that our models can adapt in a few-shot way, especially when there are fewer instances per epoch. For 64k instances/epoch models, our results suggest that they are still sample efficient, but with no significant improvement of the final performance.

The curves also show that the more data we have, the better and faster is the convergence of the models. The asymptotic performance between the two models gets close to each other by augmenting the number of instances per epoch. However, the gap between TL and NO-TL models is more significant when we have less training data, which means that in this setting, we can achieve better average tour lengths by considering a pre-trained model and transfer learning.

For our last metric, training on 50 epochs showed that for 16k and 32k instances/epoch, transfer learning models outperform NO-TL models while for the 64k instances/epoch models, it improves the learning process by making the model sample efficient. Another interesting observation can be made if we look at the performance in previous epochs. For example, in Figure 2.5-(b), the TL model achieves the performance of the NO-TL model in only 25 epochs, which is only half the number of total epochs.

For the rest of the experiments, we train models with 32k instances per epoch, as they are a good trade-off between training time and average tour lengths achieved after 50 epochs.

Diff size – In this setting, we use a pre-trained TSP model trained on instances of sizes that are different from the CVRP instances used to train our CVRP model. In this case, we distinguish two scenarios: (i) the pre-trained TSP model is trained on instances with sizes smaller than the CVRP instances and (ii) the pre-trained TSP model was trained on instances with sizes larger than the CVRP instances.

We report results on two experiments: a VRP20 model trained on CVRP instances with 20 nodes using a TSP50 pre-trained model and a VRP50 model trained on CVRP instances with 50 nodes using a TSP20 pre-trained model.

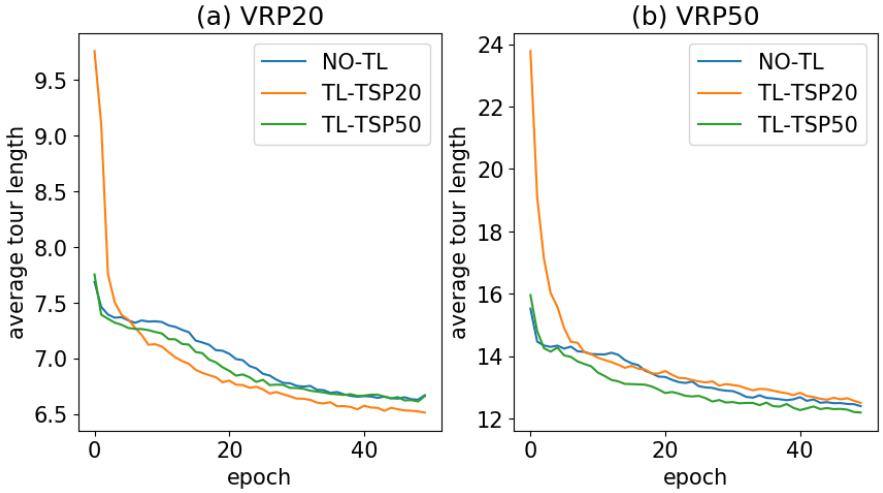


Figure 2.6: Average tour lengths for CVRP models, (a)- VRP20 and (b)-VRP50, trained without Transfer Learning (NO-TL/blue), using transfer learning with pre-trained TSP20 model (TL-TSP20/orange) and transfer learning with pre-trained TSP50 model (TL-TSP50/green).

Figures 2.6-a, 2.6-b show the evolution of the average tour length for VRP20 and VRP50, respectively. We observe that using a pre-trained TSP model trained on instances with the same size as the CVRP instances brings better average tour lengths. In Figure 2.6-a, for the VRP20 model, we can see that using a pre-trained TSP50 model (TL-TSP50) gives a jump-start performance similar to the model with no Transfer Learning (NO-TL) and better than the model that uses the TSP20 pre-trained model (TL-TSP20). In phase 2 (from epoch 5 to 30), we can see that Transfer Learning brings an improvement in average tour lengths in both settings (curves orange and green are both under the blue curve). However, we can notice that the average tour lengths are better when we use a TSP20 pre-trained model. In phase 3 (epoch 30 to 50), the model with a TSP20 pre-trained model is still better and leads to a faster learning and brings better asymptotic performance. Surprisingly, at this phase, the TL-TSP50 model behaves exactly as the NO-TL

model with no significant difference in asymptotic performance between the two models. In the VRP50 case (Figure 2.6-b), we observe that using a TSP20 pre-trained model gives worse jump-start performance than using a TSP50 model or without Transfer Learning at all. The asymptotic performance is on par with the NO-TL model, while being slightly in favor of the latter.

Our findings suggest that using a pre-trained model trained on TSP instances bigger than the CVRP instances is more efficient than using a pre-trained model trained on smaller instances than the CVRP instances. This assertion is based on the observation that there exists a region in phase 2 in Figure 2.6-(a) where the TL-TSP50 model behaves better than not using pre-training. Especially if we have a limited time budget to train (number of epochs), it is still beneficial to use transfer learning. In any case, it does not harm the learning process, as similar asymptotic performances are obtained in the worst case. Based on our results, it appears that the intuitive expectation of using a TSP50 model to pre-train for VRP20 leading to a better improvement in the learning process is not supported. Similarly, the assumption that using a TSP20 for pre-training for VRP50 would be superior to starting learning from scratch does not hold true, based on our findings. We can suppose that while learning to solve the TSP on a specific instance size, the size of these instances is somehow encoded in the learned policy and that it influences its behavior.

Diff dist – Training a CVRP model using instances generated from a different distribution than the distribution used to generate the TSP instances for the pre-trained models. Our pre-trained TSP models were trained on instances generated from a uniform distribution. Clients' coordinates of the CVRP instances have, however, been generated using a truncated normal distribution so that all city coordinates are inside the unit square. The demands are generated in the same way as in the uniform case. Figures 2.7-a and 2.7-b show a comparison between average tour lengths of models trained from scratch (NO-TL) and with a pre-trained model (TL) with respectively 20 and 50 nodes.

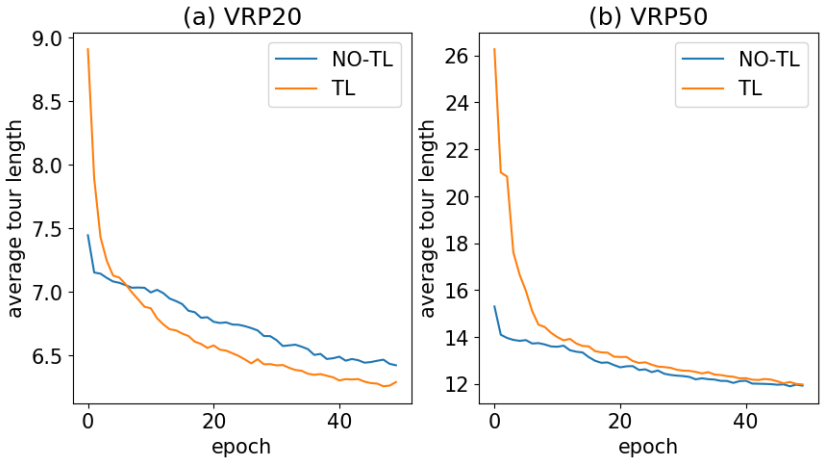


Figure 2.7: Average tour lengths per epoch between CVRP models trained without Transfer Learning (NO-TL/blue) and CVRP models trained using transfer learning (TL/orange) with 20 nodes [(a) VRP20] and 50 nodes [(b) VRP50].

Surprisingly, as we can see in Figure 2.7-a, using a pre-trained TSP20 model to train a VRP20 model brings an improvement in terms of speed of learning even if they are trained on instances coming from different distributions. The asymptotic performance using transfer learning is better than the performance without using it. The TL model achieves the performance of NO-TL in approximately half the epochs. Monitoring the jump-start performance, we can observe that neither the VRP20 nor the VRP50 can do zero-shot transfer. While the VRP20 adapts in a few-shot way, the VRP50 struggles to do so. Figure 2.7-b shows that the pre-training does not bring significant improvement for learning a representation that would help tackle the VRP50, at least for a 50 epochs training. Indeed, in this case, the asymptotic performance is on par with the NO-TL model. We can hypothesize that the size of the problem to tackle makes the learning process more difficult. There is probably a bias induced in the learning process of TSP50 from the instances' distribution (how the cities are spread in the unit square) that is not helpful for solving the VRP50, in the case where the CVRP instances are drawn from a different distribution. This bias may be less significant when the size of the instances is small.

TL as pretext task – Finally, we explore the use of TSP as a pretext task. To this end, we pre-train two models for instances of sizes 20 and 50 with CVRP instances whose features are used to solve a TSP. These pre-trained models are then trained to solve CVRP20 and CVRP50 respectively. The models are trained on 50 epochs, with 32k instances per epoch for both pretext and target tasks. The resulting models trained using TSP as a pretext task are compared with models trained from scratch and using the standard transfer learning (with a TSP trained on 1.28M data per epoch). Figures 2.8-(a), (b) show the evolution of the average tour lengths per epoch for the CVRP20 and CVRP50 respectively. In the case of CVRP20 (Figure 2.8-(a)), we observe that using the TSP as a pretext task gives a model that is more sample efficient than the model obtained using standard transfer learning in the first 20 epochs. However, it converges more slowly, and achieves a worse asymptotic performance compared with the standard transfer learning setting. Using TSP as a pretext task is also more sample efficient than training from scratch (NO-TL). On the other hand, for the CVRP50 experiments (Figure 2.8-(b)), the results are not in favor of using the TSP as a pretext task, as it has worse asymptotic performance than even training from scratch. Nevertheless, let us note that the TL-pretext model has a better jump-start performance than the other models, and is even more sample efficient in the first 5 epochs.

To explain the results obtained in this case, we investigate the final performance of the models when pre-training on TSP. Figure 2.9 shows the evolution of the average tour lengths per epoch, when using 32k CVRP instances per epoch to train TSP models with 20 cities (plot (a)) and 50 cities (plot (b)). The final performance is 4.04 for the TSP20 model, and 9.45 for the TSP50 model. These models have a gap of 5.20% and 62.37% respectively to the final performance of the TSP20 and TSP50 models that fully converged. This gap can be explained by the difference in number of instances used to train the two models (32k instances for pretext and 1.28M for fully converged models). Thus, the fact that we used a model that did not fully converge in the case of TL-pretext explains the performances observed when performing transfer learning. The CVRP20 model has decent performance compared with training on CVRP from scratch because the model's final performance gap on TSP is small. However, for the CVRP50 model, it has worse performance because it did not fully converge when pre-trained on

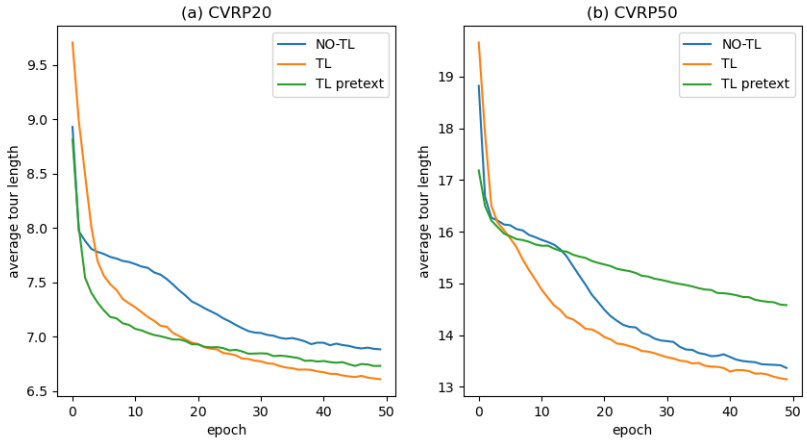


Figure 2.8: Average tour lengths per epoch for CVRP models trained without Transfer Learning (NO-TL/blue), CVRP models trained using transfer learning (TL/orange) from a TSP model pre-trained on 1.28M instances per epoch, CVRP models trained using transfer learning (TL-pretext/orange) with TSP model pre-trained on 32k instances per epoch, with instances of (a) 20 nodes and (b) 50 nodes.

TSP. Still, we observe that it has a contribution to the performance of the CVRP50 model on the first few epochs, but it led to a worst local optimum, in terms of asymptotic performance.

2.6 Conclusion and perspectives

In this chapter, we presented an empirical evaluation of transfer learning under the Neural Combinatorial Optimization framework. We illustrated this evaluation on two routing problems: the TSP and the CVRP. We presented working hypothesis and derived research questions that guided our study throughout the chapter. We also identified the metrics to monitor transfer learning from our literature review: the jump-start and asymptotic performances and performance under a fixed number of training epochs.

Several training settings were studied by varying the instances' distribution, the instances' sizes of the pre-trained models, and the number of data used in both pre-training and downstream task. We observe that, transfer learning from TSP (i)

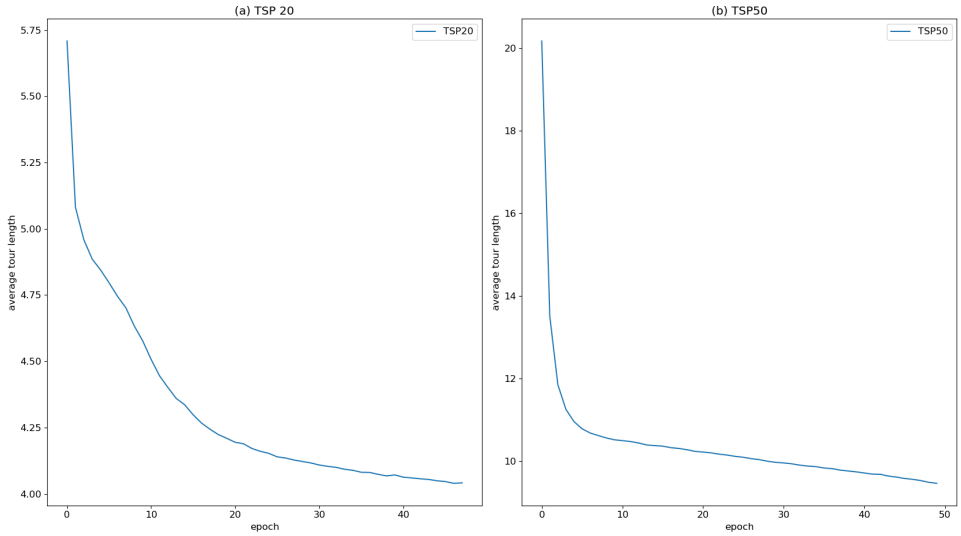


Figure 2.9: Evolution of average tour length per epoch when training on TSP with 20 cities (left) and 50 cities (right) with 32k instances per epoch.

accelerates the learning process even with relatively few instances, as it improves the sample efficiency of the pre-trained models that are trained on the CVRP and (ii) improves the results (tour lengths) compared with a model trained from random weights by achieving better asymptotic performance. We also identify the limit on the number of data per epoch, where transfer learning brings improvement for 32k instances/epoch. Our results also stress that in the worst settings, transfer learning does not harm the learning process, as asymptotic performances similar to those obtained by models with no pre-training are achieved. We also note that results vary depending on the size of the instances used in the pre-training phase: better models are obtained using models pre-trained on TSP instances of the same size as the CVRP instances. Finally, the pre-trained model used for transfer learning should be a model that was trained until full convergence. Indeed, using transfer learning pre-training from a pretext task gives results correlated with the final performance on the pretext task; the better the performance on the pretext task, the better the performance on the target task.

Future work is required to investigate how to create models that are both efficient and agnostic to the size of instances. Furthermore, training on CVRP instances sampled from a different distribution than the TSP instances proved to be

more challenging when using a pre-trained model; it does work for small instances but struggles to scale to bigger instances. We encourage future contributions to investigate this specific aspect of transfer learning for neural combinatorial optimization applied to CVRP. Indeed, if we can pre-train and train models on TSP and CVRP instances sampled from different distributions, this would lead to significant improvements since it would be possible to pre-train on synthetic instances to next efficiently train the model on real-world CVRP instances. Finally, future research directions should investigate multitask learning, which considers learning tasks simultaneously [205]. For example, we could consider learning with the same deep neural network the CVRP and the VRPTW. Intuitively, grouping the two types of instances increases the total number of available instances for training the deep neural network, which could possibly make the training process more sample efficient and give better asymptotic performance than training on both problems separately.

Chapter 3

Neural order-first split-second approach for the vehicle routing problem

In this chapter, we introduce a two-steps method based on route-first split-second strategy for the Capacitated Vehicle Routing Problem. Our approach is based on a combination of deep neural networks and an optimal split algorithm. We first present a concise literature review on two-steps algorithms, and we detail the *Split* algorithm. We then introduce our approach, along with the details of the deep neural networks used to tackle the problem. We conduct extensive computational experiments on a dataset of randomly generated instances using different search strategies. We compare our results with construction-based heuristics and two-steps heuristics, as well as state of the art metaheuristics. Furthermore, the proposed method is also evaluated on CVRPLib benchmarks. We also investigate (i) the effect of features used and (ii) the depth of the neural network on the convergence of the models, as well as (iii) the effect of the choice of neural network encoder on the overall performance of the model.

3.1 Two-steps approaches for routing problems

Two-steps approaches for the VRP have long been known in the literature. As highlighted in Chapter 1, VRPs involve a bin-packing problem as a means to determine clusters of clients, and a traveling salesman problem to order each cluster into a minimum cost route.

3.1.1 Cluster-first Route-second algorithms

Cluster-first Route-second algorithms were among the first two-steps approaches, because of their intuitiveness. In a first step, a set of feasible clusters are determined, i.e., clients are grouped so that the vehicle's capacity is not violated (Figure

3.1-(a)). In the second step, the order of visit of clients is set on using a TSP resolution method (Figure 3.1-(b)). Tyagi [174] proposed an algorithm that uses the nearest neighbor criterion in order to add clients to clusters until the vehicle's capacity is reached. Gillet and Miller [64] introduced the Sweep algorithm. It calculates the angle between each client location and the depot, and sorts the clients in ascending order based on the angle. Then, they are assigned to routes, without transgressing the vehicle's capacity. If a client cannot be assigned to a route due to the capacity constraint, a new route is created. The algorithm iterates until all clients are visited. An extension of this algorithm, called the Petal algorithm, proposes to build several routes, and then select the routes that are included into the final solution by solving a set partitioning problem. The Fisher and Jaikumar [57] algorithm determines the clusters by solving a generalized assignment problem.

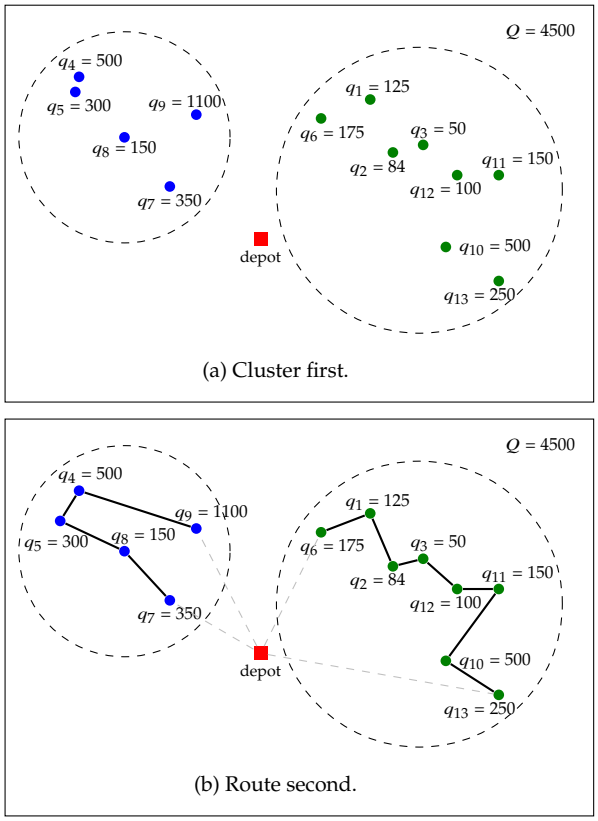


Figure 3.1: Example of a CVRP instance solved using the Cluster-First Route-Second algorithm.

3.1.2 Order-first Split-second algorithms

Order-first Split-second algorithms first order the clients into a sequence called giant tour (Figure 3.2-(a)), then, decompose it into a set of feasible routes (Figure 3.2-(b)). Beasley [17] was the first to propose the Route-first Cluster-second algorithm (RFCS) for the CVRP. He observed that the routes can be optimally extracted from the giant tour by solving a standard shortest path problem on an acyclic graph in $O(n^2)$ (n being the number of clients). RFCS uses TSP heuristics to generate a giant tour. It uses 2-OPT moves on a random permutation of clients visit order to create a giant tour. Then, it builds the routes using Floyd-Warshall algorithm [190], as detailed hereafter.

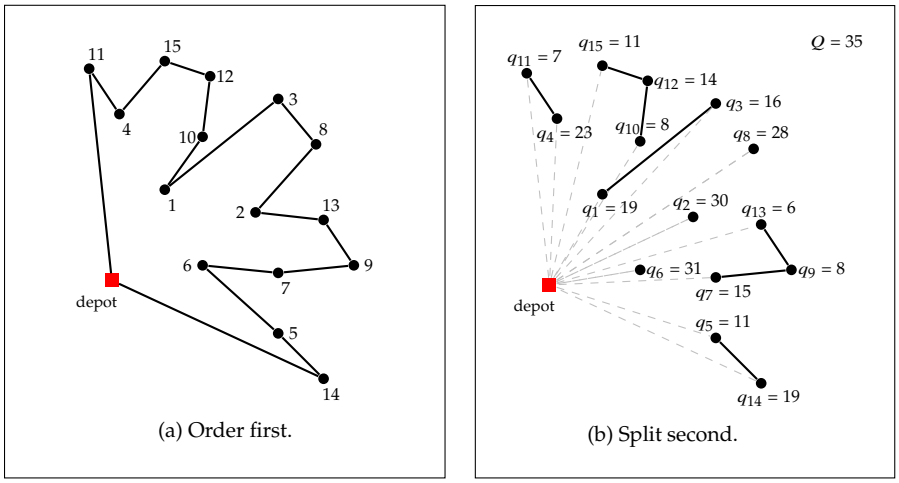


Figure 3.2: Example of a CVRP instance (P-n16-k8) solved using the Order-First Split-Second method.

After building the giant tour $\mathcal{Y} = (y_0, y_1, ..., y_n)$ with $y_0 = 0$ being the depot and $y_1, ..., y_n$ being the index of the clients, we define an auxiliary graph $H(V^H, E^H, D^H)$ with $|V^H| = n+1$. The nodes in V^H indicate the depot, either for return or departure. The edge set indicates all possible routes that start from y_i to y_j $(y_i, y_{i+1}, ..., y_j)$ that do not transgress the vehicle's capacity constraint. We formulate it as follows: $E^H = \{(i, j) \in V^H \times V^H; \quad i < j, \quad \sum_{k=i+1}^{j} q_{y_k} \leq Q\}$. The edges are weighted as follows: for an edge $(i, j) \in E^H$, we associate the total travelled distance starting from the depot to the client y_{i+1}, visiting the tour $(y_{i+1}, ..., y_j)$ and going back to

the depot from y_j:

$$D^H = \{d_{ij} = dist(0, y_{i+1}) + \sum_{\substack{k=i+1 \\ j-i>1}}^{j-1} dist(y_k, y_{k+1}) + dist(y_j, 0), \quad \forall (i,j) \in E^H\}$$

where $dist(\cdot,\cdot)$ is the function used to compute the Euclidean distance.

By construction, the auxiliary graph is a direct acyclic graph. Using a shortest path algorithm, such as Bellman's algorithm or Floyd-Warshall algorithm, we can find the shortest path from the node 0 to the node n in this graph. The associated shortest path cost represents the best solution length (total travelled distance) for the CVRP instance regarding the given giant tour.

To illustrate how the auxiliary graph is built and how to find the optimal split, we use the previous example of Figure 3.2. The represented giant tour in Figure 3.2-(a) is $\mathcal{Y} = [0, 11, 4, 15, 12, 10, 1, 3, 8, 2, 13, 9, 7, 6, 5, 14]$, with 0 being the depot, and the clients ranging from 1 to 15. The vehicle's capacity is $Q = 35$ and the demands are visible in Table 3.1. The distances between all the clients and the depot are given in Table 3.2.

With an instance of 15 clients, we build an auxiliary graph with 16 nodes, i.e, $V^H = \{0, \cdots, 15\}$. The edge set is then computed, as described above. For example, the edge $(0,1)$ is admitted because $\sum_{k=0+1}^{1} q_{y_k} = q_{11} = 7 \leq 35$, while the edge $(0,3)$ is not in the edge set because $\sum_{k=0+1}^{3} q_{y_k} = q_{11} + q_4 + q_{15} = 7 + 23 + 11 = 41 > 35$. Finally, the associated total traveled distance set D^H is computed for each edge in E^H. For the edge $(0,1)$, it corresponds to going back and forth between the depot and client $y_1 = 11$, i.e. $d_{0,1} = dist(0,11) + dist(11,0) = 56$. Another less trivial example is the total traveled distance associated with the edge $(2,5)$, which is computed as follows: $d_{2,5} = dist(0, y_3) + dist(y_3, y_4) + dist(y_4, y_5) + dist(y_5, 0) = dist(0,15) + dist(15,12) + dist(12,10) + dist(10,0) = 30 + 6 + 10 + 21 = 67$.

client id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
demand	19	30	16	23	11	31	15	28	8	8	7	14	6	19	11

Table 3.1: Clients' demands in the instance P-n16-k8.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0.0	14	21	33	22	23	12	22	32	32	21	28	30	29	31	30
1	14	0.0	12	19	12	24	12	19	21	27	7	19	16	21	33	17
2	21	12	0.0	15	22	16	11	9	12	15	11	29	19	9	24	23
3	33	19	15	0.0	21	31	25	23	8	24	12	25	9	17	37	16
4	22	12	22	21	0.0	36	24	30	26	37	12	7	13	30	44	9
5	23	24	16	31	36	0.0	13	8	25	13	26	43	35	16	8	39
6	12	12	11	25	24	13	0.0	10	23	20	16	31	26	17	21	28
7	22	19	9	23	30	8	10	0.0	18	10	19	37	28	9	15	32
8	32	21	12	8	26	25	23	18	0.0	17	15	32	17	10	31	23
9	32	27	15	24	37	13	20	10	17	0.0	25	44	31	7	16	37
10	21	7	11	12	12	26	16	19	15	25	0.0	19	10	18	34	13
11	28	19	29	25	7	43	31	37	32	44	19	0.0	16	37	51	10
12	30	16	19	9	13	35	26	28	17	31	10	16	0.0	24	43	6
13	29	21	9	17	30	16	17	9	10	7	18	37	24	0.0	21	30
14	31	33	24	37	44	8	21	15	31	16	34	51	43	21	0.0	47
15	30	17	23	16	9	39	28	32	23	37	13	10	6	30	47	0.0

Table 3.2: The distance matrix associated with the P-n16-k8 instance.

Figure 3.1.2 depicts the resulting auxiliary graph, with edges weighted with the couple (route demand, route distance). Selecting the best routes comes down to solving a shortest path on this graph from node 0 to node 15, using the routes distances as weights. The red arcs in Figure 3.1.2 represent the resulting solution using Bellman's algorithm. Thus, we can read from this solution the associated CVRP solution, and its cost. There are 8 arcs in the shortest path solution, which implies that there are 8 routes. We can deduce the CVRP solution from the shortest path solution, as follows: for a selected arc (i, j) in the shortest path solution, the clients that are in a single route in the CVRP solution are $\{y_{i+1}, y_{i+2} \cdots, y_j\}$.

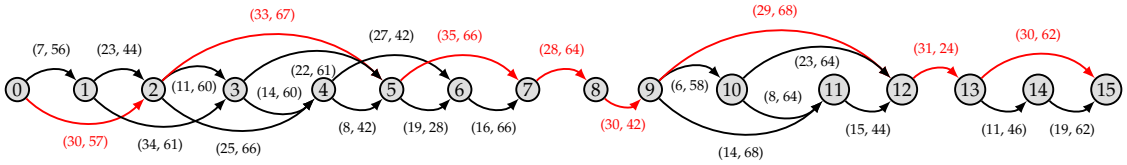


Figure 3.3: Example of an auxiliary graph of the instance (P-n16-k8) using the giant tour of the Figure 3.2-(a) and the corresponding CVRP solution using the Split algorithm. For each edge $(i, j) \in E$ the associated weights represent the total route demand $\sum_{k=i+1}^{j} q_{y_k}$, and the total traveled distance from $d_{ij} \in D$. The arcs in red correspond to the shortest path from the node 0 to the node 15, using the distances d_{ij} as weights.

This gives us the following routes:

$$\texttt{R\#1} : y_1, y_2$$
$$\texttt{R\#2} : y_3, y_4, y_5$$
$$\texttt{R\#3} : y_6, y_7$$
$$\texttt{R\#4} : y_8$$
$$\texttt{R\#5} : y_9$$
$$\texttt{R\#6} : y_{10}, y_{11}, y_{12}$$
$$\texttt{R\#7} : y_{13}$$
$$\texttt{R\#8} : y_{14}, y_{15}$$

The corresponding sequence with trip delimiters is, therefore:

$$\mathcal{Y}' = [0, 11, 4, 0, 15, 12, 10, 0, 1, 3, 0, 8, 0, 2, 0, 13, 9, 7, 0, 6, 0, 5, 14, 0]$$

The cost corresponds to the sum of the selected arcs' weights, i.e., $57 + 67 + 66 + 64 + 42 + 68 + 24 + 62 = 450$.

Let us note that, in practice, this auxiliary graph is not explicitly generated.

Instead, a dynamic programming algorithm, called *Split* [149], is directly used to extract the best solution from the giant tour, as depicted in Algorithm 5. The algorithm takes as input the giant tour $\mathcal{Y}$ (with $\mathcal{Y}_0$ being the depot), the distance matrix, the clients' demands and the vehicle capacity. It returns the cost of the optimal split of the giant tour, which corresponds to the cost of the CVRP candidate solution. *minlong* is a one dimensional array that stores at index j the cost of a shortest path from the depot to the node $\mathcal{Y}_j$. $minlong[0]$ is associated with the depot, for which the cost is 0, the rest of the array indices have an infinite cost. Then we have two nested loops with variables i and j which correspond to the subsequence of clients $(\mathcal{Y}_i, \ldots, \mathcal{Y}_j)$. We compute the load and cost of the subsequence. If the total load does not exceed the vehicle's capacity and if there is an improvement in the cost of the shortest path from the depot to the node $\mathcal{Y}_j$, then, we store the cost of the route in $minlong[j]$. The cost of the CVRP solution can be retrieved from $minlong[n]$ at the end of the execution of the algorithm.

A recent survey identifies more than 70 research papers that propose heuristics and metaheuristics that successfully tackle VRPs using the Order-First Split-Second approach [150]. Prins proposed the first genetic algorithm for the CVRP that relies on the Order-first Split-second approach, which was competitive with the best metaheuristic at that time (Tabu Search) [149]. In their approach, the author proposed a representation of the chromosomes as giant tours; thus a chromosome is a permutation of clients' visit order. They introduced the *Split* procedure, a Bellman-based dynamic programming algorithm, to extract the best routes. The resulting CVRP solution cost is then returned as a fitness evaluation of the giant tour. The giant tour representation along with the *Split* procedure were later used in a plethora of algorithms for various VRP variants. To name a few, Vidal et al. [183] used a hybrid genetic algorithm to tackle different variants of the VRP with time windows. More recently, Vidal [181] proposed a Hybrid Genetic Search algorithm dedicated to the CVRP (HGS) which achieves state-of-the-art performances on the problem.

Many reasons contribute to the success of this approach, in particular, the fact that it is computationally less expensive to build a giant tour, and then to split it, than building clusters of clients. In addition, searching for a solution among permutations of clients' visit order significantly reduces the size of the search space,

compared with the one with trip delimiters.

Algorithm 5: The Split algorithm

Inputs: giant tour $\mathcal{Y}$, number of clients n, vehicle's capacity Q, distance matrix $d_{ij} \quad \forall i, j \in \{0, \cdots, n\}$, clients demands $q_i \quad \forall i \in \{1, \cdots, n\}$
Output: $minlong[n]$ `// cost of the CVRP tours`
$minlong[0] \leftarrow 0$
for $i \leftarrow 1$ *to* n **do**
    $minlong[i] \leftarrow \infty$
end
for $i \leftarrow 1$ *to* n **do**
    $load \leftarrow 0$
    $j \leftarrow i$
    $c_1 \leftarrow \mathcal{Y}[i]$
    repeat
        $c_2 \leftarrow \mathcal{Y}[j]$
        $load \leftarrow load + q_{c_2}$
        if $j = i$ **then**
            $cost \leftarrow d[0, c_2] + d[c_2, 0]$
        else
            $cost \leftarrow cost - d[c1, 0] + d[c_1, c_2] + d[c_2, 0]$
        end
        if $load \leq Q$ **then**
            if $minlong[i-1] + cost < minlong[j]$ **then**
                $minlong[j] \leftarrow cost + minlong[i-1]$
            end
            $j \leftarrow j + 1$
        end
        $c_1 \leftarrow c_2$
    until $j > n$ *or* $load > Q$
end

This search space reduction does not make the optimal solution unattainable, since there is an *optimal* giant tour which corresponds to the optimal solution. This

can be easily verified. Indeed, if we suppose that we have an optimal solution with trip delimiter, removing them would result in a giant tour, which, if split, would give us again the optimal solution. Furthermore, thanks to the split approach, we can implicitly consider numerous feasible vehicle routes, and from them pick an optimal set of routes. This ensures to prevent too many poor quality solutions from appearing often, provided we have a good quality giant tour. Finally, the split procedure is relatively fast to execute, which makes it possible to evaluate several giant tours, and to shift the resolution process to finding an efficient method that generates the best giant tour. In the rest of this chapter, we will study the use of the Order-First Split-Second approach under the Neural Combinatorial Optimization (NCO) framework, which we will name Neural Order-First Split-Second (NOFSS).

3.2 The Neural Order-First Split-Second approach

We saw in Chapter 1 that most of the Neural Combinatorial Optimization approaches are construction-based strategies [132, 100, 103, 144]. As a reminder, the candidate solution is built by iteratively selecting the next client to visit, or the depot to refill, until all the demands are satisfied. The action to perform at each construction step is chosen based on a probability distribution that will be estimated by a deep neural network, either using supervised or reinforcement learning. This discrete probability distribution defines the probability that an extension of the partial solution under construction, considering each available choices (unsatisfied clients and depot), will lead to the optimal solution. Using such an approach, the models handle both clients routing and returns to the depot. In this context, choices of when to return to the depot are critical. Indeed, more returns to the depot can *de facto* lead to candidate solutions with a number of tours greater than the optimal one. This will result in models failing to learn interesting resolution strategies efficiently, i.e., routing *policies* because of poor quality candidate solutions, and/or large computational costs inducing prohibitive learning process (millions of learning steps). For example, a model may decide to put each client on a separate tour, when there is enough capacity in the vehicle to satisfy more requests. Among other things, solutions of this type may appear in the first few

iterations:

$$[0, y_1, 0, y_2, 0, y_3, 0, \cdots]$$

which is, usually, far from the optimal one.

On the other hand, we have seen in the previous section that successful heuristics and metaheuristics are obtained by considering the giant tour representation of the solution, while the solution and its cost are retrieved via the *Split* algorithm. Inspired by this problem decomposition, we propose the Neural Order-First Split-Second (NOFSS) approach as a novel two-steps learning-based approach which:

1. Learns how to order clients into a giant tour;

2. Optimally split the giant tour using the *Split* algorithm.

NOFSS is a generic approach that we introduce and test in the context of CVRP, even if it may be used for a larger class of routing problems. Its efficiency stems into two keys aspects. The first one is the use of an Encoder-Decoder deep neural network architecture based on (i) graph neural networks for computing the representations of the CVRP instances in the embedding space, and (ii) a GRU memory cell to encode the giant tour sequence at the time of its construction. Thus, our neural network implicitly learns to solve VRP instances by exploring the space of giant tours.

The second key aspect is the use of the *Split* algorithm to compute the total travelled distance of the CVRP solution corresponding to the giant tour generated from our deep neural network. *Split* acts as an oracle that evaluates the quality of the giant tour, and provides feedback to the deep neural network. This oracle replaces the usual NCO oracle which for a given solution with trip delimiters, i.e., $[y_0, y_1, y_2, \cdots, y_T]$, with T being the length of the solution, computes the total traveled distance is $\sum_{i=0}^{T-1} dist(y_i, y_{i+1})$. Thus, using *Split*, it is still possible to train a NOFSS model via reinforcement learning.

As previously stated, with this approach, a deep neural network can learn different policies depending on the variant of the VRP (e.g., VRP with Time Windows) without additional adaptation. The *Split* algorithm will handle the additional constraints to extract the best feasible solution from the giant tour representation and its corresponding cost, and the neural network learns the policy accordingly.

It is also worth mentioning that unlike the other learning-based construction approaches that build a solution in a variable number of steps due to the return to the depot to refill (with a number of iterations equal to twice the number of clients in the instance, in the worst case[1]), the NOFSS approach builds a giant tour in a fixed number of steps equal to the number of clients in the instance. Algorithm 6 presents the training loop, which corresponds to the REINFORCE with Rollout baseline algorithm, adapted to the NOFSS method, which we detail hereafter.

The deep neural network with parameters θ is trained over a set of I CVRP instances, for E epochs. The training dataset is split into batches of B instances. Thus, training for a single epoch requires a total of $T = \frac{I}{B}$ iterations. For a given instance $X \sim \mathcal{D}$ of n clients, our neural network defines a stochastic policy that outputs the probability of generating a sequence $\mathcal{Y}$ as a giant tour. Using the probability chain rule, with θ being the parameters of the neural network, this probability is defined as the product of the conditional probabilities of extending the giant tour with a client at step t, i.e.,

$$P_\theta(\mathcal{Y}|X) = \prod_{t=1}^{n} p_\theta(y_t|y_0, \cdots, y_{t-1}, X)$$

We define the loss as the expected tour lengths of the giant tours evaluated by the *Split* algorithm, i.e.

$$\mathcal{L}(\theta) = \mathbb{E}_{X\sim\mathcal{D},\mathcal{Y}\sim P_\theta(.|X)}\big[\text{Split}(\mathcal{Y}, X)\big]$$

The objective is to find the best parameters θ that will output good quality sequences $\mathcal{Y}$ that would result in short tour lengths. For this, we rely on gradient descent to update the parameters θ during training, by using the Adam optimizer [96]. To compute the gradient of the loss, we use the REINFORCE with baseline algorithm. REINFORCE links the gradient of the loss to the gradient of the log of the probabilities output by the deep neural network [192]:

$$\nabla_\theta \mathcal{L}(\theta) = \mathbb{E}_{X\sim\mathcal{D},\mathcal{Y}\sim P_\theta(.|X)}\Big[\Big(\text{Split}(\mathcal{Y}, X) - b(X)\Big)\nabla_\theta \log P_\theta(\mathcal{Y}|X)\Big]$$

[1]The worst case corresponding to routing each client in a single route.

The gradient $\nabla_\theta \mathcal{L}(\theta)$ is approximated using Monte Carlo sampling over a batch of B i.i.d CVRP instances as follows:

$$\nabla_\theta \mathcal{L}(\theta) \approx \frac{1}{B} \sum_{i=1}^{B} \left[\Big(\text{Split}(\mathcal{Y}_i, X_i) - b(X_i) \Big) \nabla_\theta \log P_\theta(\mathcal{Y}_i | X_i) \right]$$

The baseline $b(X)$ is used to reduce the gradient variance, leading to an acceleration of the learning process. We use the greedy rollout baseline, which uses a delayed copy of the policy network to generate a giant tour $\mathcal{Y}^{BL}$ by a greedy search strategy; the client with the highest probability of appearance is added to the giant tour at each step. This giant tour baseline is then evaluated using the *Split* algorithm, i.e. $b(X) = \text{Split}(\mathcal{Y}^{BL}, X)$. This baseline turned out to be more efficient than actor-critic or REINFORCE with an exponential moving average baseline [100]. During validation, if the performance of θ is significantly better than that of θ^{BL} according to a t-test ($\alpha = 5\%$), the baseline is updated with the parameters θ (lines 14–16 in Algorithm 6).

Algorithm 6: NOFSS REINFORCE with Rollout Baseline

Inputs: initial policy network parameters θ, number of epochs E, batch size B, number of instances I, number of clients n, vehicle capacity Q, t-test threshold α

$T \leftarrow \frac{I}{B}$

$\theta^{BL} \leftarrow \theta$

for $e \leftarrow 1$ *to* E **do** // train for E epochs

  for $t \leftarrow 1$ *to* T **do** // loop over the T instance batches

    // Get a batch of B CVRP instances with n clients

    $X_i \leftarrow \text{getInstance}(n, Q), \quad \forall i \in \{1, \cdots, B\}$

    // Sample a giant tour according to the learning policy P_θ

    $\mathcal{Y}_i \leftarrow \text{SampleGiantTour}(X_i, P_\theta), \quad \forall i \in \{1, \cdots, B\}$

    // Generate a giant tour greedily according to the policy $P_{\theta^{BL}}$

    $\mathcal{Y}_i^{BL} \leftarrow \text{GreedyGiantTour}(X_i, P_{\theta^{BL}}), \quad \forall i \in \{1, \cdots, B\}$

    // Evaluate giant tours total travel cost

    $L_i \leftarrow \text{Split}(X_i, \mathcal{Y}_i, Q) \quad \forall i \in \{1, \cdots, B\}$

    $L_i^{BL} \leftarrow \text{Split}(X_i, \mathcal{Y}_i^{BL}, Q) \quad \forall i \in \{1, \cdots, B\}$

    $\nabla_\theta \mathcal{L} \leftarrow \frac{1}{B} \sum_{i=1}^{B} (L_i - L_i^{BL}) \nabla_\theta \log P_\theta(\mathcal{Y}_i | X_i)$ // Compute the loss

    $\theta \leftarrow AdamW(\theta, \nabla_\theta \mathcal{L})$ // update neural network parameters

  end

  if *t-test*$(P_\theta, P_{\theta^{BL}}) < \alpha$ **then**

    $\theta^{BL} \leftarrow \theta$

  end

end

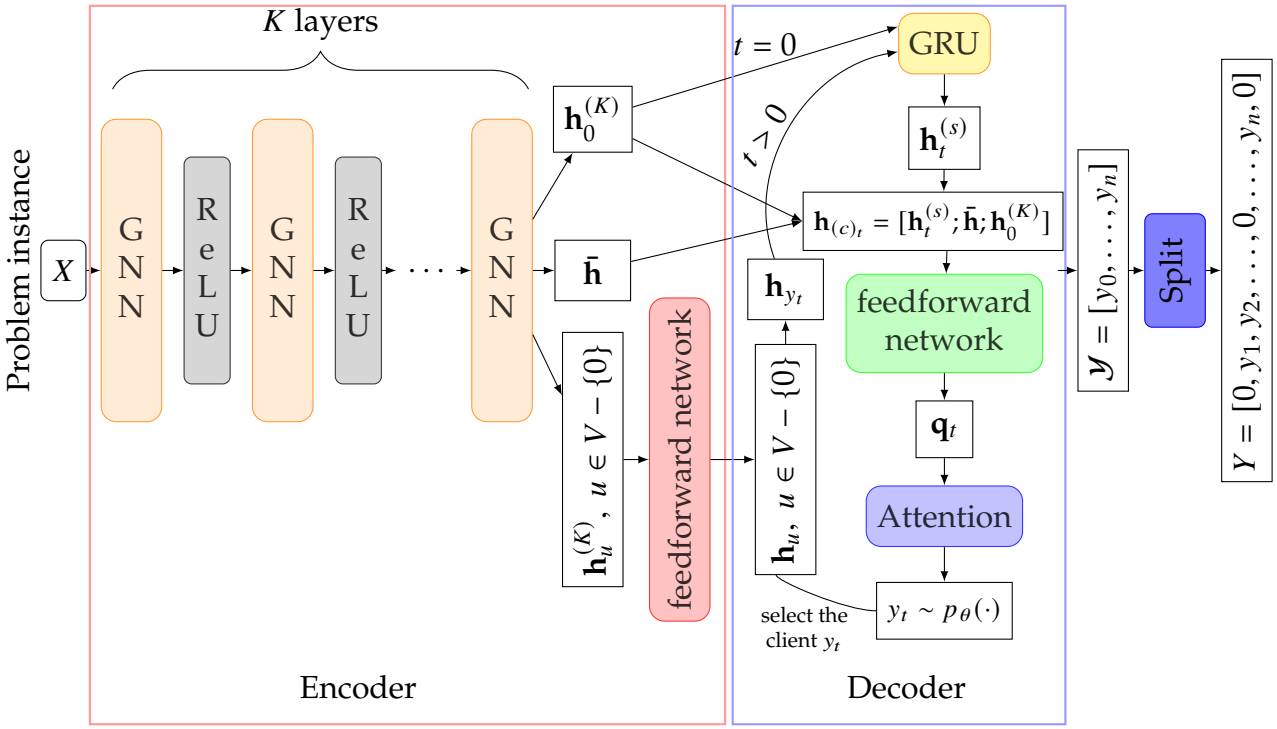


Figure 3.4: Proposed NOFSS model for solving CVRP instances.

3.3 The deep neural network architecture

The NOFSS approach is agnostic to the choice of encoding and decoding components of the model. Thus, we propose to train various encoder-decoder models that rely on different graph neural networks (GNNs) encoders and a GRU recurrent cell for decoding. Figure 3.4 depicts our proposed encoder-decoder architecture that we detail in the subsection 3.3.2. We first detail the features that we will use as inputs to our deep neural networks in the following subsection.

3.3.1 Instance features

Although deep neural networks' power lies in their automatic feature extraction, providing good initial features can improve the learning process and reduce training time. In general, only raw features are given as input, and from this, the neural

network computes a new representation. However, in the CVRP case, we find it useful to compute features that may result in a better representation instead of only using the raw features (node coordinates and demand). Let $G(V, E)$ be the graph associated with the CVRP instance, with V being the nodes and E being the edges that connect each pair of nodes. For each instance, we define the nodes and edges features as follows.

Node features. Each node $u \in V$ is represented as a quadruplet $(x_u, y_u, \hat{q}_u, a_u)$, where $(x_u, y_u) \in [0, 1]^2$ are the node coordinates, $\hat{q}_u = \frac{q_u}{Q} \in [0, 1]$ is the normalized demand and $a_u = atan\left(\frac{y_u - y_0}{x_u - x_0}\right) \in]-\frac{\pi}{2}, \frac{\pi}{2}[$ is the polar angle between the node u and the depot node 0. Generally, in the literature, only the triplet (x_u, y_u, q_u) is used, but we find it convenient to add information about the polar angle between a client and the depot. Computing the polar angle does not add any computational overhead and was already a characteristic of interest in handcrafted heuristics, such as the Sweep algorithm [64].

Edge features. For each edge $(u, v) \in E$, we define the edge features as the Euclidean distance between the nodes u and v ($e_{u,v} = \sqrt{(x_u - x_v)^2 + (y_u - y_v)^2}$, $\forall (u, v) \in E$). The distance between two nodes in the instance is an interesting feature in the case of VRPs because it appears in the objective function as the quantity of interest to be minimized.

3.3.2 NOFSS encoding-decoding architecture

3.3.2.1 Encoding

Since VRPs have inherently a graph theory formulation, we propose to use graph neural networks for encoding the problem instances into an embedding space. The literature of GNNs is rich, which yields to a plethora of choices for the encoder. Since there is no certainty about the best GNN encoder, we experiment three of them for our approach: GCN (a spectral GNN)[97], GAT (a spatial GNN) [179] and TransformerConv (a spatial GNN) [163]. In all cases, the encoders are made of K similar GNN layers with a ReLU activation function in between. Their inputs

consist of node and edge features, and their outputs are node embeddings for both the clients and depot $\mathbf{h}_u^{(K)} \in \mathbb{R}^d$, $\forall u \in V$ and a graph embedding computed using an average pooling layer $\bar{\mathbf{h}} = \frac{1}{|V|} \sum_{u \in V} \mathbf{h}_u^{(K)}$. Finally, to distinguish the clients embeddings from the depot embedding $\mathbf{h}_0^{(K)}$, we use a linear transformation of the client's embeddings: $\mathbf{h}_u = \mathbf{W}_c \, \mathbf{h}_u^{(K)} + \mathbf{b}_c, \forall u \in V - \{0\}$, with $\mathbf{W}_c \in \mathbb{R}^{d \times d}, \mathbf{b}_c \in \mathbb{R}^d$ being respectively the weights and the bias of the layer.

Each type of GNN differs from the other in the way it computes the node embeddings. GCN computes the node embeddings as a linear transformation of the weighted sum of the embeddings of each node's neighbors, including itself (Equation 3.1). The weights $\alpha_{u,v}$ correspond to the normalized edge values $e_{u,v}$ (Equation 3.2).

$$\mathbf{h}_u^{(k)} = \mathbf{W}^{(k)} \sum_{v \in \mathcal{N}_u \cup \{u\}} \alpha_{u,v} \, \mathbf{h}_v^{(k-1)}, \; k \in \{1, \ldots, K\} \tag{3.1}$$

$$\alpha_{u,v} = \frac{e_{u,v}}{\sqrt{|\mathcal{N}_u| \; |\mathcal{N}_v|}} \tag{3.2}$$

with $\mathcal{N}_u, \mathcal{N}_v$ being the neighborhood sets of nodes u and v, respectively, $\mathbf{W}^{(k)} \in \mathbb{R}^{d \times d}$ the weight matrix of the k^{th} GNN block, and $\mathbf{h}_v^{(k-1)} \in \mathbb{R}^d$ the node embedding from the $(k-1)^{th}$ GNN block.

On the other hand, GAT proposes to take advantage of the improvements introduced by the Transformer neural networks. Firstly, it uses the multi-head mechanism to compute L different embeddings for each node. Each head computes a node representation as a weighted sum of the embeddings of the node's neighbors, including itself (Equation 3.3). The way the weighting is done differs from GCN. In GAT, we use the attention mechanism similar to the additive attention to assign a higher weight to more important neighboring nodes (Equation 3.4). The edge features are included in the attention mechanism only. The final node embedding is computed by concatenating the heads' outputs, followed by an activation function (Equation 3.8).

$$\mathbf{h}_u^{(k),l} = \sum_{v \in \mathcal{N}_u \cup \{u\}} \alpha_{u,v}^l \cdot \mathbf{W}^{(k),l} \mathbf{h}_v^{(k-1),l}, \ l \in \{1, \ldots, L\}, k \in \{1, \ldots, K\} \tag{3.3}$$

$$\alpha_{u,v}^l = \underset{v \in \mathcal{N}_u \cup \{u\}}{softmax}(LReLU(\mathbf{a}^\intercal [\mathbf{U}^{(k),l}; \mathbf{V}^{(k),l}; \mathbf{E}^{(k),l}])), \ l \in \{1, \ldots, L\}, k \in \{1, \ldots, K\} \tag{3.4}$$

$$\mathbf{U}^{(k),l} = \mathbf{W}^{(k),l} \mathbf{h}_u^{(k-1),l}, \ l \in \{1, \ldots, L\}, k \in \{1, \ldots, K\} \tag{3.5}$$

$$\mathbf{V}^{(k),l} = \mathbf{W}^{(k),l} \mathbf{h}_v^{(k-1),l} \ l \in \{1, \ldots, L\}, k \in \{1, \ldots, K\} \tag{3.6}$$

$$\mathbf{E}^{(k),l} = \mathbf{W_e}^{(k),l} e_{u,v} \ l \in \{1, \ldots, L\}, k \in \{1, \ldots, K\} \tag{3.7}$$

$$\mathbf{h}_u^{(k)} = ReLU\left(\left[\mathbf{h}_u^{(k),1}; \ldots; \mathbf{h}_u^{(k),L}\right]\right), \ k \in \{1, \ldots, K\} \tag{3.8}$$

with $\mathbf{W}^{(k),l} \in \mathbb{R}^{d \times d}, \mathbf{W_e}^{(k),l} \in \mathbb{R}^{1 \times d}$ are the weight matrices of the l^{th} head of the k^{th} GNN block for the node and edge embeddings respectively, $\mathbf{h}_v^{(k-1),l} \in \mathbb{R}^{d/L}$ is the node embedding from the l^{th} head of the $(k-1)^{th}$ GNN block, and $\mathbf{a} \in \mathbb{R}^d$ is a vector of learned parameters.

Finally, TransformerConv is similar to GAT, since both use the multi-head and the attention mechanisms. They differ in the way they aggregate the neighbor's embeddings to compute node embeddings. TransformerConv uses different neural network parameters $(\mathbf{W_1}, \mathbf{W_2})$ to encode the updated node and its neighbors. Moreover, the corresponding weight of the updated node is equal to 1, while the neighbors weights are computed using an attention mechanism. TransformerConv uses the scaled dot-product attention to compute the nodes embeddings. The edge features are integrated similarly in both the aggregation and the attention mechanism. The final node embedding is obtained likewise as in GAT, by using an activation function on the result of the concatenation of the output of the heads.

$$\mathbf{h}_u^{(k),l} = \mathbf{W_1}^{(k),l}\mathbf{h}_u^{(k-1),l} + \sum_{v \in \mathcal{N}_u} \alpha_{u,v}^{l} \cdot \left(\mathbf{W_2}^{(k),l}\mathbf{h}_v^{(k-1),l} + \mathbf{E}^{(k),l}\right),\ l \in \{1,\dots,L\}, k \in \{1,\dots,K\} \tag{3.9}$$

$$\alpha_{u,v}^{l} = \underset{v \in \mathcal{N}_u \cup \{u\}}{softmax}\left(\frac{\mathbf{U}^{(k),l}\left(\mathbf{V}^{(k),l} + \mathbf{E}^{(k),l}\right)^{\intercal}}{\sqrt{d}}\right),\ l \in \{1,\dots,L\}, k \in \{1,\dots,K\} \tag{3.10}$$

$$\mathbf{U}^{(k),l} = \mathbf{W_3}^{(k),l}\mathbf{h}_u^{(k-1),l},\ l \in \{1,\dots,L\}, k \in \{1,\dots,K\} \tag{3.11}$$

$$\mathbf{V}^{(k),l} = \mathbf{W_4}^{(k),l}\mathbf{h}_v^{(k-1),l},\ l \in \{1,\dots,L\}, k \in \{1,\dots,K\} \tag{3.12}$$

$$\mathbf{E}^{(k),l} = \mathbf{W_e}^{(k),l}e_{u,v},\ l \in \{1,\dots,L\}, k \in \{1,\dots,K\} \tag{3.13}$$

$$\mathbf{h}_u^{(k)} = ReLU\left(\left[\mathbf{h}_u^{(k),1};\dots;\mathbf{h}_u^{(k),L}\right]\right),\ k \in \{1,\dots,K\} \tag{3.14}$$

with $\mathbf{W}_i^{(k),l} \in \mathbb{R}^{d \times d}, \quad i \in \{1,\dots,4\}$ are the weight matrices of the l^{th} head of the k^{th} GNN block for the node embeddings, $\mathbf{W_e}^{(k),l} \in \mathbb{R}^{d \times 1}$ is the weight matrix of the l^{th} head of the k^{th} GNN block for the edge embeddings, $\mathbf{h}_v^{(k-1),l} \in R^{d/L}$ is the node embedding from the l^{th} head of the $(k-1)^{th}$ GNN block, and $\mathbf{a} \in \mathbb{R}^d$ is a vector of learned parameters.

3.3.2.2 Neighborhood definition

In its canonical form, instances of the CVRP are represented by complete graphs. There is, *a priori*, no way to assert that an edge connecting two nodes will not be in the optimal solution. However, using a complete graph as input for the GNNs may result in the appearance of the over-smoothing phenomenon; the representations of the nodes in the graph end up being the same, as the number of GNN layers increases [134]. To prevent this, we propose, *for the encoding phase only*[2], to define for each node its neighborhood. Previous works successfully used this trick for the TSP. For example, Khalil et al. [94] defined as fixed

[2]during decoding, we still consider a complete graph.

number of neighbors ($\kappa = 10$) for each node, regardless of the size of the graph. Other works consider the $n \cdot \kappa\%$ nearest neighbors, so that the node degrees scale better on small and large instances. However, for the CVRP, we argue that this neighborhood definition does not consider the instance's characteristics. Thus, we propose to use a different strategy to set the neighborhood of a client and the depot. We define the neighborhood $\mathcal{N}_u$ of a client node $u \in V - \{0\}$ as the κ nearest nodes in terms of Euclidean distance union the depot 0, i.e., $\mathcal{N}_u = \{v_1, v_2, ..., v_\kappa \in V; \|v_1 - u\| \leq \|v_2 - u\| \leq ... \leq \|v_\kappa - u\|\} \cup \{0\})$. We choose to connect all the clients to the depot because each one of the clients can be a potential candidate from which the vehicle travels back to the depot. For the depot, we consider that it is connected to every client, since every client can be a candidate for the departure from the depot. An example of an instance neighborhood definition is depicted in Figure 3.5. The central node (red square) represents the depot, while the other nodes (blue circles) represent the clients. An edge exists between nodes u and v if $v \in \mathcal{N}_u$. In addition, we propose to compute the number of nearest neighbors κ, based on the characteristics of the instance in terms of the number of clients, their demands, and the capacity of the vehicles instead of selecting an arbitrary number of them. We set it to be the average number of clients per route as if they were uniformly distributed on the routes, i.e. $\kappa = \frac{n}{m}$ with n being the number of clients and m being the lower bound of the number of routes ($m = \left\lceil \frac{\sum_{i=1}^{n} q_i}{Q} \right\rceil$). Of course, such a neighborhood definition does not correspond to the number of clients in each route in the optimal solution, nor the number of routes itself. However, it offers the advantage of being individualized per instance, which is, in our view, better than setting an arbitrary number for κ.

3.3.2.3 Decoding

In the deep learning jargon, producing a giant tour corresponds to decoding a sequence of clients' order. Since the ordering of the clients is important in this phase, we use a GRU recurrent cell to capture the sequence's representation $\mathbf{h}_t^{(s)}$ at each step t [31]. Decoding happens iteratively, in n steps. At $t = 0$, we only use the depot representation $\mathbf{h}_0^{(K)}$ as input of the GRU. For $t > 0$, the GRU takes as input the previously selected client representation at step $t - 1$ and the depot

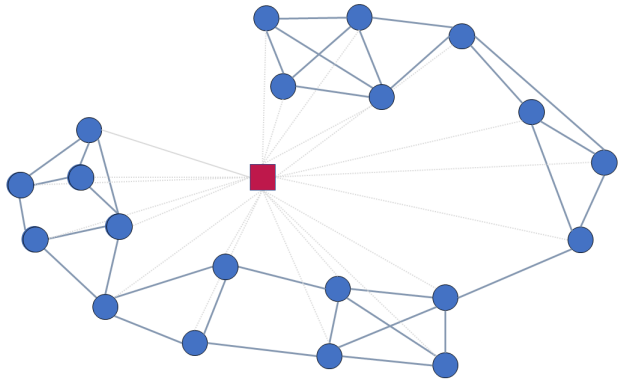


Figure 3.5: CVRP instance with relationships between neighboring nodes (central square node is the depot).

representation $\mathbf{h}_0^{(K)}$. This results into a vector $\mathbf{h}_t^{(s)}$ that represents the giant tour under construction, i.e.,

$$\mathbf{h}_t^{(s)} := \begin{cases} \text{GRU}([\mathbf{h}_0^{(K)}]), & t = 0 \\ \text{GRU}([\mathbf{h}_{y_{t-1}}, \mathbf{h}_0^{(K)}]), & t > 0 \end{cases}$$

The graph embedding $\bar{\mathbf{h}}$, the depot embedding $\mathbf{h}_0^{(K)}$ and the sequence embedding $\mathbf{h}_t^{(s)}$ are then concatenated together to form a context vector $\mathbf{h}_c \in \mathbb{R}^{3d}$. The context vector is then passed to a feedforward layer made of two linear layers with the *ReLU* activation function in between to output a query vector $\mathbf{q} \in \mathbb{R}^d$ i.e.,

$$\mathbf{q} = \mathbf{W}_2 \; ReLU(\mathbf{W}_1 \; \mathbf{h}_c + \mathbf{b}_1) + \mathbf{b}_2$$

with $\mathbf{W}_1 \in \mathbb{R}^{d \times 3d}$, $\mathbf{W}_2 \in \mathbb{R}^{d \times d}$, $\mathbf{b}_1, \mathbf{b}_2 \in \mathbb{R}^d$ being the parameters of the feedforward network.

To compute the probability of selecting the next client $p_\theta(y_t | y_0, ..., y_{t-1}, X)$, we compute attention scores s_u $(\forall u \in V - \{0\})$ using a scaled dot-product with a masking mechanism to avoid selecting the same client twice. These scores are

then clipped within $[-C, C]$ using $tanh$ [20].

$$s_u = \begin{cases} C \cdot tanh\left(\frac{\mathbf{q}_t \cdot \mathbf{h}_u^\top}{\sqrt{d}}\right), & u \neq y_{t'} \quad t' < t, C = 10 \\ -\infty & \text{otherwise} \end{cases}$$

The attention scores are converted into a probability distribution using the softmax function

$$p_\theta(y_t = i | y_0, ..., y_{t-1}, X) = softmax(s_i)$$

By setting the value of the attention score to $-\infty$, we can perform the masking of already visited clients. Thus, when passed to the softmax function, its associated probability will be 0.

3.4 Experimental protocol

3.4.1 Data generation

We follow the data generation protocol of Nazari et al. [132] to consider 3 types of CVRP instances with number of clients $n = 20$, 50 and 100. For each problem size, we have generated $100k$ instances for training, and two sets of $10k$ instances for validation and test. Clients and depot locations are uniformly generated from $\mathcal{U}\left([0, 1]^2\right)$. The clients' demands are also uniformly drawn from $\{1, \ldots, 9\}$. Vehicles' capacities are set to 30, 40 and 50 respectively for $n = 20$, 50, 100.

3.4.2 Hyperparameters

We use an embedding dimension $d = 128$ and a uniform parameter initialization for our deep neural networks $\mathcal{U}\left([-\frac{1}{\sqrt{d}}, \frac{1}{\sqrt{d}}]\right)$ and set the learning rate to $\eta = 10^{-3}$. The models are trained with a time limit of 100 hours and batch size $B = 128$. For each encoder type, we use $K = 3$ GNN blocks. GAT and TransformerConv use $L = 8$ heads for the multi-head attention mechanism.

3.4.3 Baselines

We use HGS[3] [181] as baseline as it is one of the state-of-the-art metaheuristics for the CVRP. We also use classical CVRP heuristics[4]: (i) RFCS [17] as a two-steps Order-first Split-second heuristic, (ii) Sweep [64] as a two-steps Cluster-first Route-second approach, and (iii) Nearest Neighbor heuristic as a single-step construction approach [176]. We also trained the model with TransformerConv encoder in end-to-end (FL-Transformer), i.e., the model manages the visit of the clients and the return to the depot (FL stands for Full Learning).

We first notice that NOFSS models are faster to train, completing $E = 1000$ of learning epochs in the 100 hours time budget, while the Full-learning models perform 1000, 500 and 200 training epochs for instance sizes of 20, 50 and 100 respectively. The validation is done using a greedy decoding which considers the highest probability at each decoding step, for both the policy and the baseline models. For the exploitation of the learned policies on the test set (inference), we use greedy decoding and a sampling strategy which samples 1280 candidate solutions for each test instance from the probability distributions given by the models.

Table 3.3 reports the results of each approach on the test set, specifying: average solution lengths (obj.), the average gap (in percentage) to the best average solution lengths, and the running time (in seconds) to output a candidate solution for a single instance.

3.5 Computational results

[3] https://github.com/vidalt/HGS-CVRP
[4] https://github.com/yorak/VeRyPy

Method	$n = 20$			$n = 50$			$n = 100$		
	obj.	gap (%)	time (s)	obj.	gap (%)	time (s)	obj.	gap (%)	time (s)
HGS	6.13	0.00	0.003	10.34	0.00	0.09	15.57	0.00	0.69
RFCS	6.85	11.74	0.16	11.80	14.11	0.79	17.94	15.22	3.00
Sweep	7.55	23.16	0.01	15.60	50.93	0.06	28.56	83.37	0.23
Nearest neighbor	7.39	20.57	0.0004	12.63	22.19	0.001	18.95	21.68	0.01
NOFSS-GCN (G)	6.83	11.41	0.0008	12.31	19.05	0.003	19.41	24.66	0.007
NOFSS-GAT (G)	6.59	7.50	0.006	11.74	13.53	0.02	18.34	17.80	0.05
NOFSS-Transformer (G)	6.50	6.03	0.006	11.57	11.89	0.02	18.13	16.44	0.06
FL-Transformer (G)	6.49	5.87	0.006	11.34	9.67	0.02	17.69	13.61	0.06
NOFSS-Transformer (S)	6.24	1.79	1.37	11.03	6.67	1.56	17.45	12.07	2.43
FL-Transformer (S)	6.18	0.81	2.09	10.79	4.35	2.35	17.32	11.23	8.29

Table 3.3: NOFSS vs. other algorithms. FL for Full-Learning; exploitation, greedy (G), sampling (S).

3.5.1 Comparison with an end-to-end construction model

Figure 3.6 presents the evolution of the average solution length per epoch during training and validation on CVRP instances with 20 clients (left) and 50 clients (right)[5]. During training, candidate solutions are sampled from the model and their total lengths are averaged over the training set. Let us note that the models' parameters are updated each time a batch is processed via gradient descent, thus the performance of the models changes for every batch during training, while validation is performed using the model resulting from the processing of the last batch in the training set, which is supposed to be the best model achieved at the end of the epoch. Furthermore, in validation, we use a greedy decoding instead of sampling. The evolution of the average solution lengths shows that the NOFSS model can learn an implicit policy for solving the CVRP by learning to output an indirect representation of the solution (giant tour).

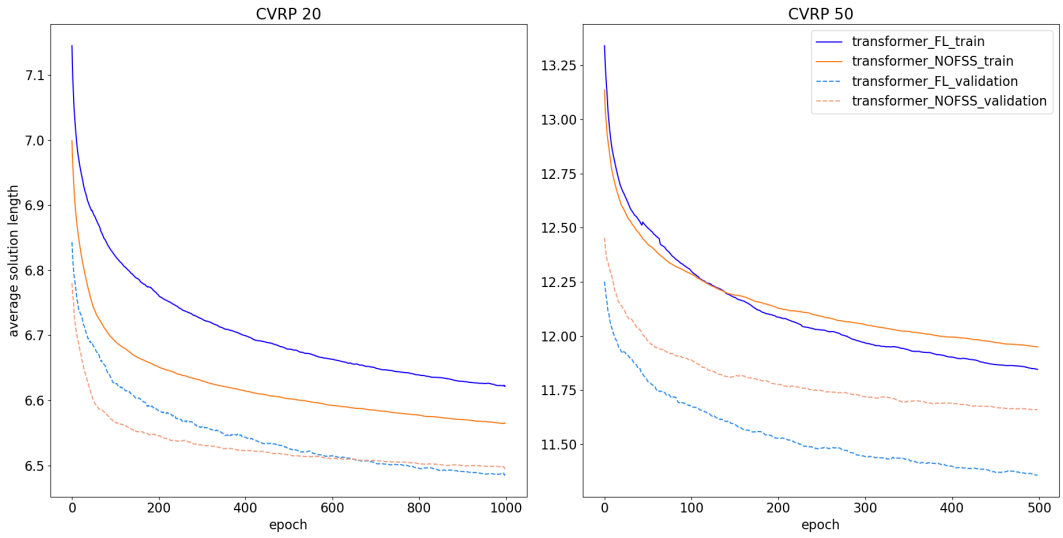


Figure 3.6: Learning curves in training and validation for Full-learning (blue) and NOFSS models (orange) on CVRP instances with 20 (CVRP20) and 50 clients (CVRP50); lower is better.

On instances with 20 clients, we can observe that during training, the NOFSS

[5]We observe the same evolution in the plot with 100 clients.

model achieves better average solution lengths than the Full-learning model. On validation, we observe the same trend as in training, but starting from the 600^{th} epoch, the Full-learning model slightly outperforms the NOFSS model. The equivalent performance of the two models is confirmed on the test set with average solution lengths of 6.50 and 6.49 on greedy decoding for NOFSS and Full-learning respectively with similar execution times. On sampling decoding, similar performances are observed, with 0.9% difference in performance between the two models, but with an advantage in execution time in favor of NOFSS. On CVRP with 50 clients, we observe that NOFSS has a better jump-start performance on training and the Full-learning model has a better final performance. We observe 2% difference in performance for greedy and sampling decoding on the test set. We also note similar sampling times for the two types of models in greedy decoding, while NOFSS being 1.52, 1.50 and 3.41 times faster in sampling respectively for $n = 20$, 50 and 100.

3.5.2 Comparison with handcrafted heuristics

When compared with handcrafted heuristics, we can observe from Table 3.3 that either with greedy or sampling search strategies, NOFSS models outperform the Sweep, Nearest neighbor and RFCS algorithms, except for the instances with 100 clients where only the sampling strategy outperforms RCFS. Let us note that while RFCS and NOFSS belong to the same type of two-steps strategy, using a sampling strategy for NOFSS makes it surpass RFCS in both solution quality and execution time. Furthermore, the difference in average solution lengths may suggest that NOFSS's learned policy is different from a handcrafted strategy that uses TSP heuristics to generate the giant tour. We further investigate the number of test instances where NOFSS performs better than RFCS and vice versa. The results are presented in Table 3.4. It shows that overall, NOFSS with stochastic sampling outperforms RFCS for 99.29%, 98.72% and 90.57% of the instances of sizes of 20, 50, 100 respectively.

n	NOFSS better than RFCS	RFCS better than NOFSS
20	9929	71
50	9872	128
100	9057	943

Table 3.4: Summary of the number of instances where NOFSS with stochastic sampling is better than RFCS and vice versa, for different instance sizes (20, 50, 100).

3.5.3 Generalization to different instance sizes

We propose to study the generalization of the models trained on a set of instances with a specific size on instances of different size. For this, we evaluate the models trained on a specific instance size by using the different test sets with instances of size 20, 50 and 100. We use the NOFSS-Transformer model and compare it with its full learning counterpart (FL-Transformer). Table 3.5 sums up our results. We report the average solution lengths for both greedy (G) and sampling (S) search strategies. Regarding the greedy decoding, we present the results for the models trained on the different instance sizes while for stochastic sampling, we focus on the model trained on instance sizes which seems more promising based on our findings on the greedy decoding. For example, in the first line of Table 3.5, Transformer-20-(G) refers to the model that uses TransformerConv encoder, which was trained on instances with 20 clients, and evaluated on the test sets using greedy (G) decoding. The first column means that the model is the NOFSS-Transformer which is tested on instances of size 20, 50 and 100. The second column corresponds to the FL-Transformer which is tested on instances of size 20, 50 and 100.

We observe that for the Transformer-20-(G), the NOFSS model has a better generalization property than the Full-learning model, with performance similar for $n = 20$ and $n = 100$ and better for $n = 50$. Since training models on instances with 20 clients is faster, it is relevant to stress that NOFSS would therefore be a better choice.

For Transformer-50-(G) and Transformer-100-(G), it appears that, for $n = 20$ NOFSS models have better performances than their Full-learning counterparts while staying competitive for $n = 50$ and $n = 100$. An interesting result observed

on Transformer-50-(G) is its good generalization to CVRP instances with 100 clients, as it appears that it achieves better performance than the models trained on instances with 100 clients. This may suggest that relevant invariants that are beyond the instance size are learned while training on instances with 50 clients. We push further our investigations on Transformer-50-(S) by analyzing its performance using a stochastic sampling search strategy. While for the instances with 20 clients, the models stay competitive with the ones trained on that size, they achieve the best performances on the sets with instances with 50 and 100 clients. Transformer-50 appears to be a good trade-off between learning speed (it is faster to train than Transformer-100) and performance.

Trained model	NOFSS			Full-learning		
	20	50	100	20	50	100
Transformer-20-(G)	6.50	11.62	18.34	6.49	12.01	18.33
Transformer-50-(G)	6.64	11.57	17.97	6.76	11.34	17.52
Transformer-100-(G)	6.94	11.79	18.13	6.98	11.65	17.69
Transformer-50-(S)	6.31	11.03	17.40	6.25	10.79	17.22

Table 3.5: Comparison of average solution lengths achieved by the NOFSS and Full-learning models on different instance sizes of the test set using greedy search (G) and stochastic sampling (S).

3.5.4 Evaluation on CVRPLib instances

We performed an evaluation of the NOFSS model using Active Search strategy on the CVRPLib instances, for the different sets A, B, E, F, P, and X. For the X instances, because of memory consumption, we only tackled instances from 100 to 240 clients. For each instance, our model samples a total of 576,000 candidate solutions. The number of sampled candidate solutions per batch depends on the size of the instance and is depicted in Table 3.6. For small instances, until 32 nodes, we sample 16 candidate solutions per batch, which means that the algorithm performs 36,000 iterations. For bigger instances ($n > 200$), due to memory limits, we only sample 64 candidate solutions per iteration.

We compare our algorithm with handcrafted heuristics. Table 3.7 presents the average gap per set of instances. The detailed results on each instance are available in Appendix B and are summarized in the box plot proposed in Figure 3.7. These results confirm that our algorithm outperforms RFCS, Nearest neighbor, and Sweep as it achieves better gaps in all the instance sets.

The gaps found by our algorithm on these sets match the orders of magnitude of the gaps observed on the randomly generated instances, especially on sets with small size instances (A, B, E, P). For the medium size instances of the set M, the average gap is 3.71%, while for the medium to big size instances of set X, this gap becomes more important (6.49%) but is still acceptable. The box plots in Figure 3.7 detail some statistics about the gaps. For instance, we observe that 75% of the instances in the sets (A, B, E, P, M) have a solution that is under 5% gap from the best solution found by HGS, while for the set X, the third quartile is at around 8% gap. For the set F, this statistic is not relevant, since there are only 3 instances in the set. Due to their characteristics of having many clients closer to each other, these instances are more difficult to solve for our model, with an average gap of 5.60%.

number of nodes n	batch size
$n \leq 32$	16
$33 \leq n \leq 64$	32
$65 \leq n \leq 128$	64
$129 \leq n \leq 200$	128
$n > 200$	64

Table 3.6: Number of sampled candidate solutions for each instance sizes intervals.

Set	NOFSS (± std)	NCO-AM (± std)	RFCS	Nearest Neighbor	Sweep
A	**2.87** (±1.56)	6.18 (±6.12%)	14.52	23.82	47.20
B	**2.45** (±2.09)	4.42 (±1.97%)	10.50	20.43	23.10
E	**2.91** (±1.96)	3.09 (±2.83%)	14.70	24.28	48.58
F	**5.60** (±4.47)	10.16 (±4.23%)	11.78	27.26	71.92
P	**2.51** (±1.51)	2.77 (±2.09%)	14.99	22.50	42.35
M	**3.71** (±1.54)	7.38 (±3.37%)	16.76	29.44	108.14
X	**6.49** (±2.54)	90.09 (±126.93%)	15.34	18.99	108.74

Table 3.7: Comparison between the average gap per set of CVRPLib instances (%).

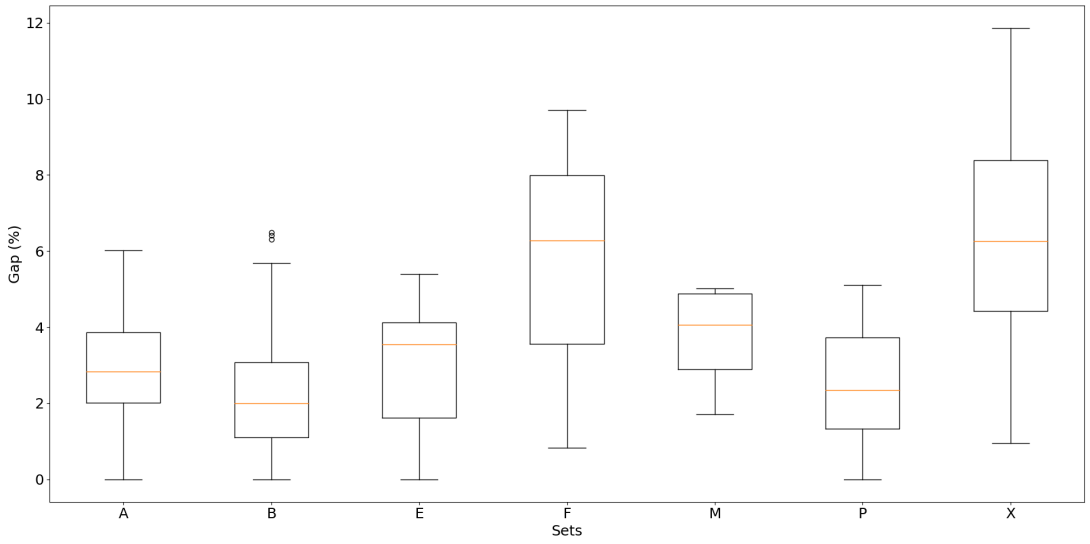


Figure 3.7: Box plot detailing the gap to HGS results of the execution of NOFSS on CVRPLib instance sets.

Regarding the execution times, we report in Table 3.8 the average of the total execution time and the average time until the model reaches its incumbent solution. As we can see, our model needs, on average, less than an hour to sample the 576,000 candidate solutions for the small size sets, while it needs an average of 1.3 to 1.8 hours for the medium to big size instances. Regarding the average time to find the best solution, it is less than 23 minutes for the small size instances of the sets (A, B,

Set	Avg. execution time (min.) (± std (min.))	Avg. time to best solution (min.) (± std (min.))
A	50.71 (±10.2)	12.94 (±12.0)
B	51.38 (±8.87)	22.88 (±17.5)
E	51.37 (±8.14)	15.33 (±14.4)
F	53.37 (±8.27)	20.77 (±18.0)
P	52.63 (±6.92)	8.59 (±9.04)
M	80.01 (±16.9)	71.55 (±22.8)
X	109.95 (±46.7)	96.74 (±47.2)

Table 3.8: Summary of average total execution time and average time until the best solution for NOFSS on CVRPLib sets.

E, F, P). We note that our model needs more time to find a good solution for sets with clustered clients (sets B and F) compared with sets with randomly distributed clients (sets A, E and P). For the medium to big size instances (set M and X), our model continues to find better solutions nearly until the end of the total execution time. This is because these instances are more difficult to solve, since the search space is larger for instances with more than a hundred clients. Moreover, these sets contain a mix of instances with randomly distributed clients and clustered clients. Thus, for some instance, the clustering property adds complexity to the size.

Finally, we compare the results of NOFSS with the ones obtained by NCO-AM in Chapter 1 (see Table 3.7). We observe that, when running the NOFSS model in Active Search algorithm, its performance is better than NCO-AM's. While they have comparable performance on the instances of sets E and P, the difference in gaps between the two models is more visible on the set X. In this case, NOFSS is able to produce better solutions that NCO-AM for the majority of the instances. Nevertheless, there exist instances where NCO-AM performs better than NOFSS, for example, the instance X-n242-k48, where NCO-AM achieves a gap to the HGS solution of 3.91% while NOFSS achieves 7.93%. This suggests that there is still room for improvement for our approach.

3.6 Model study

3.6.1 Influence of the type of encoder

We investigate the influence of the choice of GNN encoder on models' performance. Figure 3.8 shows the evolution of the average solution lengths per epoch in training and validation phases for the 3 types of GNN encoders: GCN, GAT and TransformerConv on CVRP with 20 and 50 clients. We observe the same trends for both training and validation phases, with TransformerConv having the best convergence, followed by GAT encoder and finally by GCN encoder. The instances' representation plays an important role in the resolution process because a good representation leads to the exploitation of meaningful features and, thus, gives a better solution. The choice of the encoder seems to be a critical part of the model's architecture. It appears from these results that spatial GNNs better perform than spectral GNNs in our evaluation setting. This means that the attention-based node aggregation gives a better representation than the aggregation based on node degrees, by assigning different weights to a nodes' neighborhood. While TransformerConv and GAT are both spatial GNNs, it appears that the way they exploit the node and edges information has an impact on the overall performance of the models. Possible explanations are that TransformerConv has more parameters than GAT, and uses the edge values representations in the neighborhood aggregation. Another possibility is that the dot-product attention better estimates the contributions of each neighboring node than the additive attention used in GAT.

3.6.2 On features' influence

We investigate the contribution of the use of the polar angle feature for nodes representation. For that, we trained the same model, by omitting this feature, i.e., we trained a model that considers only the coordinates and the normalized demands as nodes features. Figure 3.9 shows the evolution of the average solution length per epoch for a NOFSS model with TransformerConv encoder trained with the polar angles features (in blue) and without this feature (in orange) on CVRP with 20 clients. As we can see, adding this feature does improve the learning

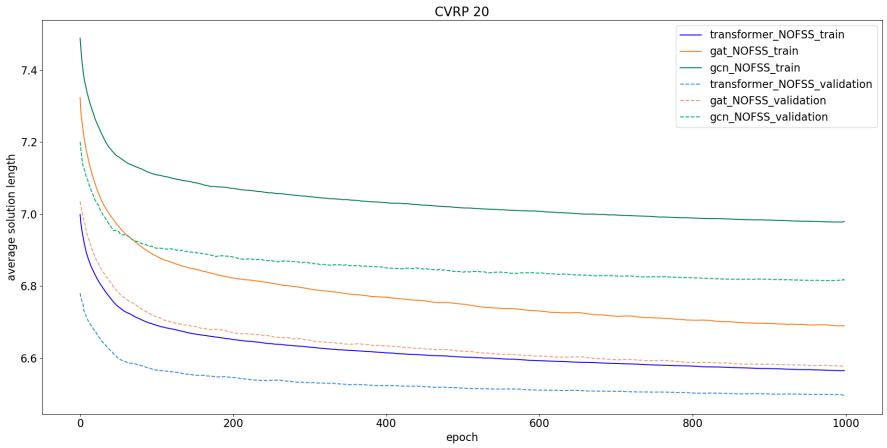


Figure 3.8: Comparison of Graph Neural Network encoders on models' performance (training and validation).

process. The model has a better starting performance. In addition, it learns faster than the model without this feature. The validation curves confirm that the model that uses the polar angles achieves better performance.

3.6.3 Using a deeper model

Our last investigation concerns the depth of the GNN encoder. Figure 3.10 shows the evolution of solution length per epoch in train and validation for a shallow model (in blue) and a deeper model (in orange). It highlights an interesting case, where deeper neural networks fail to learn better models than their shallow counterpart. As we can see, the average solution length is constant in the first hundred epochs, which means that the model cannot learn any meaningful heuristic from the data. Starting from the epoch 150, the average solution length begins to decrease, which means that the model starts to learn, but it cannot catch with the shallow model which achieves better average solution lengths in both training and validation phases. This finding suggests to us that neural networks must be carefully designed to achieve better performances, otherwise either the learning process can slow down, or, in the worst case, the model cannot recover from the region it is stuck in and completely fails to learn a meaningful heuristic.

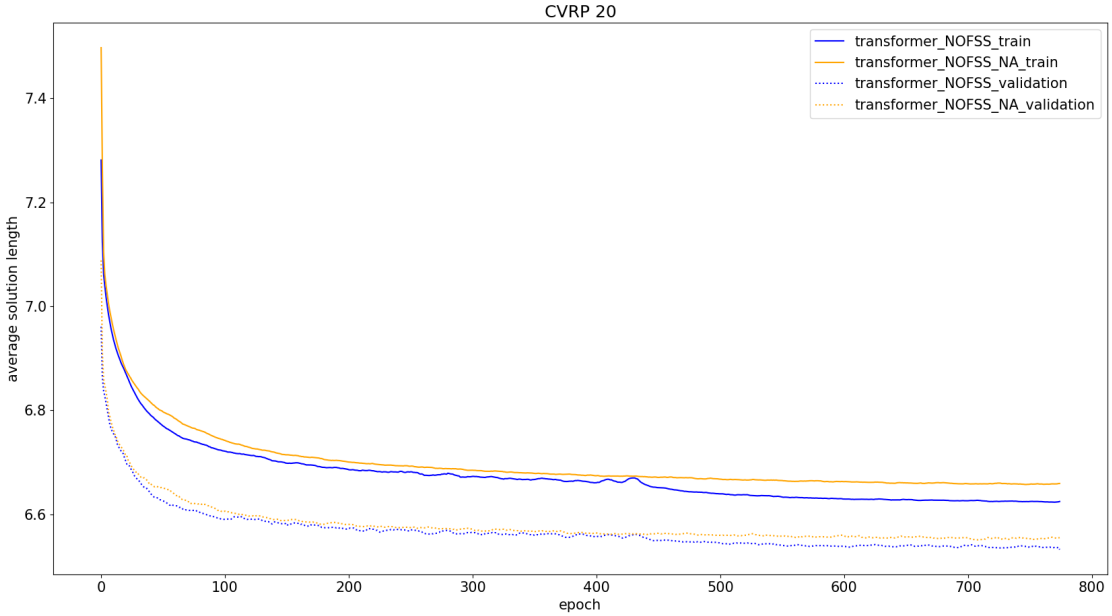


Figure 3.9: Comparison of TransformerConv NOFSS models trained with additional polar angles features (in blue) and without polar angles features (in orange), lower is better.

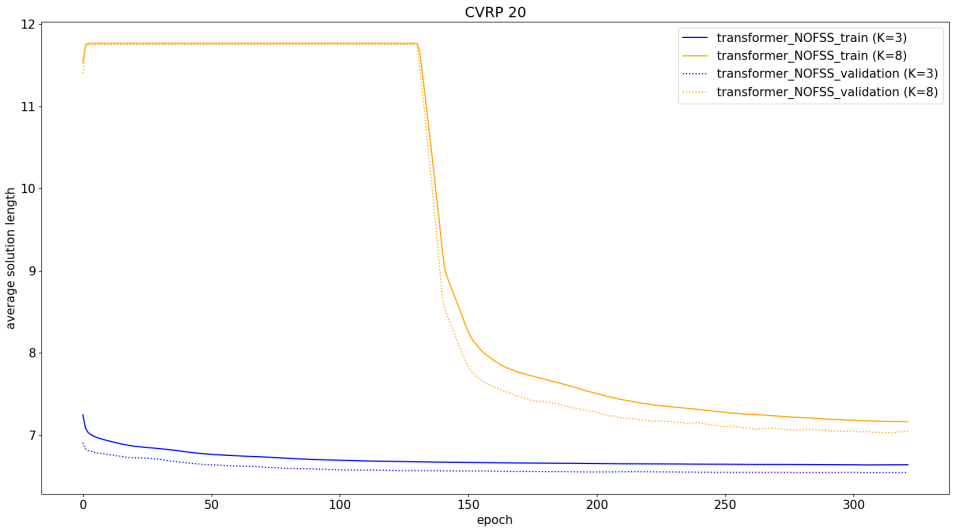


Figure 3.10: Comparison of NOFSS models with $K = 3$ TransformerConv layers (in blue) and with $K = 8$ layers (in orange), lower is better.

3.7 Conclusion and perspectives

In this work, we proposed Neural Order-First Split-Second (NOFSS), a two-steps algorithm hybridizing a deep neural network model and an exact tour splitting procedure for the Capacitated Vehicle Routing Problem (CVRP). To the best of our knowledge, this is the first model that proposes a hybridization between a deep neural network and a dynamic programming algorithm to successfully learn an implicit policy based on giant tour generation to solve the CVRP. We conducted extensive experiments on the proposed models with various Graph Neural Network encoders, which highlighted the importance of carefully designing the deep neural network for tackling the problem. We also introduced a new way to compute the number of neighboring nodes that takes into consideration the instance's characteristics. Furthermore, we proposed to add the polar angle between the clients and the depot as a node feature, and empirically observed that it improves the learning process, and the final performance of the models.

We compared our model with classic CVRP heuristics and its corresponding end-to-end version. Our results show that NOFSS outperforms classic CVRP heuristics while being competitive with its end-to-end full-learning counterpart, even if it does not outperform it. NOFSS is however faster than end-to-end approaches in both training and evaluation. It also shows good generalization properties when trained on instances with a specific size and applied to solve instances of different sizes. Besides this, the NOFSS model is easier to implement than its end-to-end alternative, while still not relying on sophisticated handcrafted search strategies to find good quality solutions. We also evaluated the performance of our approach on the CVRPLib benchmarks, and confirmed the performances on different instances sizes.

In summary, the results presented in this chapter encourage us to investigate more on the generalization of the NOFSS model to instances of bigger sizes. In addition, it would be interesting to evaluate it on other problems, such as the Vehicle Routing Problem with Time Windows. Future work will focus on the integration of this model with local search algorithms, to provide a good warm start to initiate the search.

Chapter 4

Deep reinforcement learning for the ride-hailing problem

In the previous chapters, we investigated different aspects of the application of end-to-end deep reinforcement learning on static and deterministic VRPs, with the Capacitated Vehicle Routing Problem as the default test bed. In this chapter, we shed light on the study of deep reinforcement learning methods on dynamic and stochastic VRPs. Precisely, we study the Dynamic Ride-Hailing Problem with stochastic travel times, which is a variant of the pickup and delivery problem for the transport of people. Dynamic and stochastic problems are inherently more difficult to tackle than their static and deterministic counterparts, even for handcrafted methods. Indeed, as dynamic problems require making decisions in real time, it is difficult to plan ahead. Moreover, the stochasticity of the problem adds uncertainty, which makes it difficult to predict specific outcomes. Thus, feasible solutions are bound to the realization of uncertain events. Since deep neural networks can learn complex patterns and relationships in data, and deep reinforcement learning allows accommodating to changes in real-time decision-making problems, they seem to be an interesting research direction for tackling these problems.

The problem tackled in this chapter was part of the 12^{th} DIMACS implementation challenge [101]. The competition gathered 13 teams, and 4 of them were able to submit their results on the challenge. This chapter presents our approach, which ranked 3^{rd} in the challenge. Other contributions to the challenge can be

found in the following references [4, 142].

In what follows, we first review the use of learning-based methods for ride-hailing problems. Then, we describe the problem that is studied throughout this chapter based on the description given in the DIMACS implementation challenge, and we formulate it as a reinforcement learning problem. Next, we introduce the different solution methods we designed for the problem and the deep neural network used to tackle it. This is followed by extensive computational experiments and comparisons between the different solution methods. Finally, we summarize our findings, and we give future research directions.

4.1 Literature review

The development of technologies like navigation services, real-time traffic data, and mapping, as well as the widespread adoption of the internet and smartphones contributed to the emergence and rapid adoption of ride-sharing services, such as Blablacar, Uber, Lyft. As an illustration, in the city of New York, the NYC Taxi & Limousine Commission, which is responsible for licensing and regulating taxis, reports data regarding the use of taxis and ride-sharing applications (see Figure 4.1). Through this data, we can observe that the trend has been towards the use of ride-sharing applications, with a rising trend since 2015. In 2022, these applications accounted for over 600000 trips, compared to 100000 for taxis. Similarly,

the market share of ride-sharing applications has significantly increased recently, representing over 83% in 2022 [159].

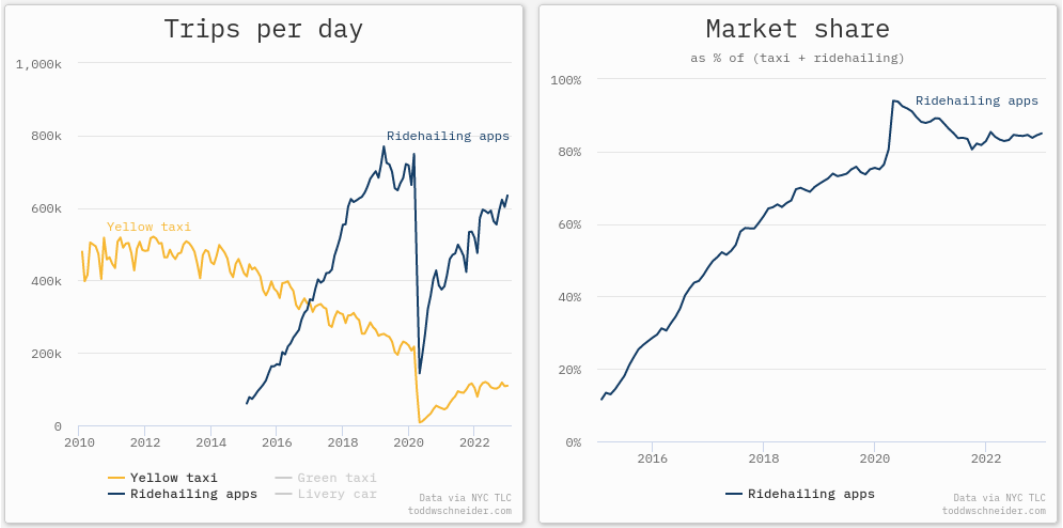


Figure 4.1: Left: Evolution of the number of trips per day from 2010 to 2022 in New York City for yellow taxis and ride-hailing applications. Right: Evolution of market share of ride-hailing applications as of % of (yellow taxis + ride-hailing) [159]

In a typical ride-sharing service, we distinguish five modules [151]:

1. Pricing: establishes the cost of a ride;
2. Matching: assigns the ride to an available vehicle;
3. Repositioning: assigns an available vehicle to an idling location;
4. Pooling: adjusts the pricing, matching, repositioning, and routing;
5. Routing: guides vehicles to different locations.

Once a user requests a ride, the first module that is called is the pricing module, which computes its cost. The user can either accept or decline the ride. Upon acceptance, the matching module assigns the ride to an available vehicle. Once the vehicle has driven the customer to their destination, the driver receives the fare and becomes available for the next ride. In cases where multiple passengers

share a ride, the pooling module adjusts the pricing, matching, repositioning, and routing. The ride-sharing application can propose to a vehicle to reposition via the repositioning module to direct the vehicle to parking areas that exhibit a high likelihood of an imminent request. Finally, the routing module is triggered to guide the vehicle to the pick-up location, destination of the ride, and idling locations.

The democratization of ride-sharing is accompanied by challenges that must be addressed by companies operating in this industry. For example, order cancellation may happen before matching the request to a vehicle, or when the vehicle is en route to pick up the customer. This complicates the matching problem. In addition, the problem of repositioning presents a significant challenge, as it involves selecting an optimal idle location that would enable the vehicle to be near a future ride request. Furthermore, the routing in ride-sharing applications differs from the standard shortest-path problem, since it must adapt to traffic condition, most often characterized stochastically [164].

The matching problem, also known as ride dispatching or order dispatching, is the subject of several studies from the reinforcement learning perspective [199, 188, 85, 209, 77, 202, 28, 102]. The rationale for using deep reinforcement learning rather than handcrafted heuristics stems from the fact that heuristics tend to prioritize immediate customer satisfaction, whereas deep reinforcement learning prioritizes the maximization of long-term rewards.

In Xu et al. [199], the matching problem is formulated as a Markov decision process, where each vehicle is considered as an agent. For each vehicle, they define the state as a pair of (time, location). The actions are either "serve a request" or "reposition in an idling location". The reward is the revenue earned by serving a request. The city is divided into regions (zones) for discretizing the action space. They consider that repositioning does not induce any cost, and perform value estimation in a discrete tabular state space, using dynamic programming. Wang et al. [188] adopt the same formulation, except that they consider a single vehicle. The problem is then solved using a model-free deep Q-learning approach (see section 1.2.2.2), using a Multi-layer perceptron (MLP) to compute the Q-values. Although trained for a single vehicle, the authors argue that the learned policy can be used in a multi-driver setting. In Zhou et al. [209], the authors propose a multi-agent reinforcement learning approach, where each vehicle acts as an independent

agent. Furthermore, they consider a reward composed of the accumulated driver income as well as the order response rate, defined as the proportion of served requests to the total orders in one day. They used a deep Q-learning with an MLP for the Q-network to optimize the agent's policy, as well as a KL-divergence to accelerate the learning process.

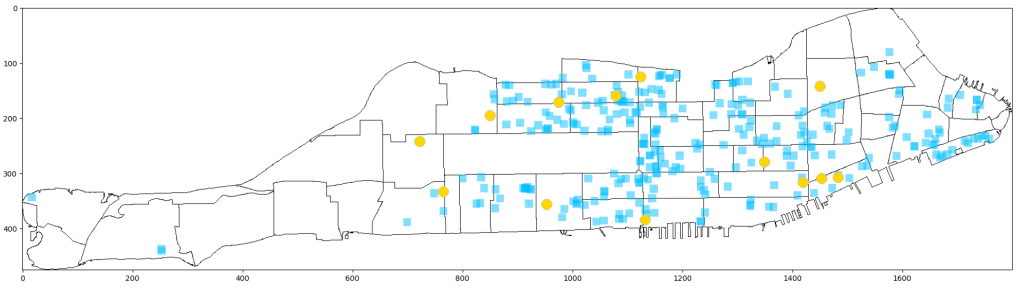


Figure 4.2: Representation of the city of Manhattan with its zoning. Yellow dots represent vehicles and blue dots represent repositioning locations.

Other works consider the joint matching and repositioning problems. In Jin et al. [85], they use a hexagonal grid-world representation of a city, where each grid is considered as an agent. Each grid (agent) handles the vehicles in its zone. The vehicles are divided into an order dispatching group and a repositioning group. Authors also model additional agents that represent districts with multiple grids inside. Moreover, they formulate the problem as a partially observable Markov decision process[1], solved using hierarchical reinforcement learning[2] [82]. They used policy gradient to train the agents with real-world data from a ride-sharing company. In Holler et al. [77] and Kullman et al. [102], the authors consider a central agent responsible for fleet management that handles both matching and repositioning tasks. Holler et al. [77] designed a deep neural network that uses two different MLPs for vehicle and request encoding. They introduce an attention mechanism to compute a global context vector, which is used in two other MLPs to compute a Q-value for repositioning and request assignment. Kullman et al. [102] address a more complex problem. In addition to the repositioning and matching

[1]A partially observable Markov decision process is used to model decision-making where the agent cannot fully observe the system's state.

[2]Hierarchical reinforcement learning decomposes a reinforcement learning problem into a hierarchy of tasks.

problems, they consider an electric fleet of vehicles. This requires addressing the extra issue of repositioning the vehicles to a charging station [102]. They considered real-world data from the city of Manhattan. In their study, the authors divided the city into various zones and utilized one-hot encodings to represent each zone (see Figure 4.2). They used a Double Deep Q-learning algorithm and a neural network similar to the one applied in Holler et al. [77] to train two agents for covering both customer and vehicle perspectives, respectively. Their findings revealed that the vehicle-centric agent performed better in terms of cumulated rewards.

Alternative studies examine the joint pricing and dispatching problem [28] or the joint pricing and repositioning problem [202]. Chen et al. [28] used a contextual bandit algorithm for the pricing problem, and temporal difference learning for vehicle dispatching [171]. Specifically, the pricing decisions are discrete changes in the percentage of the price, and are chosen using a contextual bandit algorithm that incorporates the long-term values learned by the value network into the rewards. Yuan et al. [202] formulated the pricing and the repositioning as regression problems, and they trained several models for that purpose, including two MLPs. They find out that for both tasks, their approaches based on deep neural networks best performed. Their experiment over the New York City dataset shows an improvement of 6.7% on the number of served requests compared with the original optimization approach based on Model Predictive Control optimization.

Regarding the experimental setup, the aforementioned papers utilized simulators that rely on real-world data. Several works, including [199, 188, 209, 77, 28] utilized a simulator based on DiDi ride-sharing application data. Other works considered real data from the NYC Taxi & Limousine Commission to create a simulator over the city of Manhattan [202, 102].

In what follows, we will consider a ride-hailing problem, a variant of ride-sharing, where only one passenger shares the ride with the driver. Furthermore, our work focuses on the matching and repositioning problems.

4.2 Problem definition

The problem addressed in this study shares similarities with the one presented in Kullman et al. [102], with the exception that electric vehicles are not considered,

Symbol	Meaning
V	Number of vehicles
$\mathcal{V}$	Set of vehicles
C_F	Fixed income
C_D	Per-distance income
T	Time horizon
TC	Travel cost
w^*	Maximum waiting time

Table 4.1: Symbols and associated meanings.

and therefore there are no charging-related constraints to be considered. Furthermore, the following problem definition and modeling stem from the corresponding DIMACS implementation challenge [101].

We consider the dynamic ride-hailing problem with stochastic travel times, where we have a homogeneous fleet of vehicles $\mathcal{V} = \{0, 1, ..., V-1\}$ (with V being the number of vehicles) located in idling locations, where they wait for client's requests. This fleet is controlled by a central operator, which either dispatches them to serve trip requests that arise randomly across the city of Manhattan (matching), or repositions them into other idling locations (lots) to anticipate future requests.

Objective. The goal is to define a policy for the central operator that maximizes the total profit earned over a time horizon of T hours. When serving a request, a vehicle earns a fixed income C_F and a per-distance income C_D that depends on the distance between the request's origin and destination. We also associate with each vehicle a travel cost TC that depends on the total travelled distance. This includes the distance to go pick the client at a given position, the distance to the client's destination, and the distance to an idling location. Thus, the total profit is computed as the difference between the revenue earned from serving client's and the travel costs:

$$Profit = \sum_{r \in requests} (C_F + C_D \times dist\_serving(r)) - TC \times dist\_travel_T \quad (4.1)$$

where $dist\_serving$ is a function that computes the distance between a request's origin and destination, and $dist\_travel_T$ is the total distance traveled by the vehicles over the entire time horizon T.

Stochasticity. Depending on the day of the week and hours of the day, the requests may arise in different areas of the city of Manhattan. Moreover, the central operator does not know in advance where and when requests may arise. Thus, it cannot have a predefined vehicle allocation plan, and must make decisions as requests come in. The operator must also build a strategy for deploying the fleet of vehicles to the idling locations to respond to requests in the shortest possible time. This deployment must be done intelligently so that the travel costs are not too high, but also so that the vehicles are positioned to respond quickly to client requests. In addition, to insure a quality of service, a vehicle must arrive at client's location within a fixed time of w^* minutes. After this time, if the customer is not picked up by a vehicle, the operator cancels its request. We also consider road traffic congestion in the form of stochastic travel times. The travel time is a function of the distance between the start and the end points, as well as the speed of the vehicle. This speed depends on the time slot of the customer's request location and the location of the vehicle. For example, during rush hour, depending on the vehicle's and request's geographical locations in Manhattan, the speed of the vehicle may be higher or lower. We assume that the vehicle's speed follows a normal distribution. Table 4.2 shows an example of the parameters used to sample the vehicle's speed. The mean and standard deviation speeds (in km/s) depend on:

- the elapsed time from midnight (in minutes),
- the departure zone (the zone where the vehicle is located),
- the arrival zone (the zone where the request is located).

For example, if we serve a customer from a location in zone 4 to another location in zone 4, and the elapsed time from midnight is 15 min, the speed of the vehicle for the run will be sampled from $\mathcal{N}(4.75e^{-}03, 1.24e^{-}03)$

puzone	dozone	min	speed_mean	speed_stddev
4	4	0	4.77e-03	1.26e-03
4	4	15	4.75e-03	1.24e-03
4	4	30	4.73e-03	1.23e-03
4	4	45	4.71e-03	1.22e-03
4	4	60	4.68e-03	1.20e-03

Table 4.2: Examples of vehicle's speed data based on the departure (puzone) and arrival (dozone) zones, the minutes elapsed from midnight (min), mean and standard deviation speed (speed_mean, speed_stddev).

Note that this table of parameters is not directly accessible by the central operator, making it more complex to allocate vehicles to requests.

Vehicle's job types. A vehicle can either serve a request or receive a repositioning instruction. To describe the current vehicle activity, each vehicle maintains a queue of three $''jobs''$, i.e. $jobs = [job_1, job_2, job_3]$. The possible job types for the vehicles are[3]:

- Idling (IDLE): the vehicle is waiting at a designated idling location (lot).
- Repositioning (REPO): the vehicle is on its way to a lot.
- Setup (PICKUP): the vehicle is on its way to pick up a customer.
- Processing (PROCESS): a vehicle is transporting a customer to its location.
- Null (NULL): no job specified for the vehicle.

Triggering a decision by the operator. The central operator makes decisions at a time step t when:

- a new request arises;

[3]changes have been made to the denominations used in the DIMACS challenge to facilitate the presentation.

- a vehicle goes from $jobs = [PROCESS, NULL, NULL]$ to $[NULL, NULL, NULL]$;
- MAX_TIME seconds have elapsed since the last decision step. This allows the operator to make a decision on requests that have been put on the waiting list.

Request assignment. Request assignment to a vehicle v is possible when:

- the vehicle can travel to the request's location within w^* minutes. If the expected arrival time exceeds this time constraint, then, the assignment is ignored[4];
- $job_1 \in \{NULL, IDLING, REPO\}$;
- the vehicle v has at most one assigned request: a request can be assigned to a vehicle that has $jobs(v) = [PROCESS, NULL, NULL]$, which will become $jobs(v) = [PROCESS, PICKUP, PROCESS]$. Note that it is not possible to assign a request when the vehicle is on its way to pick a customer, i.e., $jobs(v) = [PICKUP, PROCESS, NULL]$.

Furthermore, additional constraints must be respected:

- when a vehicle is processing a request and has another assigned request, it must first drop off its current customer at its destination. Then, it can begin to travel to its second customer. Thus, transitions from $jobs = [PROCESS, NULL, NULL]$ to $jobs = [PROCESS, PICKUP, PROCESS]$ do not imply a detour;
- a vehicle cannot stay in a configuration such as $jobs = [NULL, NULL, NULL]$; it must be assigned a request or reposition;
- a request is only assigned once: it is not possible to re-optimize a solution once a vehicle's dispatching plan has been decided by the operator;

[4] when a request is assigned to a vehicle, the operator also provides an idle location. Therefore, if w^* is not respected, the vehicle can reposition to an idling location, if it is not processing a request (e.g., $jobs = [NULL, NULL, NULL]$).

- assigned requests cannot be canceled, i.e., a vehicle with $jobs = [PICKUP, PROCESS, NULL]$ cannot switch to $[NULL, NULL, NULL]$;
- a vehicle that is repositioning to an idling location can be assigned to another one;
- idling vehicles will stay at their idling location until they are assigned to a request or repositioned.

We sum up all these vehicles' jobs transitions in the diagram of Figure 4.3. Blue arcs indicate the states conditions for performing request assignments, while the red ones represent the state from which it is possible to reposition. The *"start"* arrow at the node $[IDLE, NULL, NULL]$ indicates that, at the beginning, all vehicles are waiting in an idling location. The remaining arcs indicate that a vehicle that is described with the job types in the node at a time t, can be observed being in the corresponding transition node at $t + 1$. For example, a vehicle that has $jobs = [PICKUP, PROCESS, NULL]$ at a time t can be observed at $t + 1$ either at the same state, which means that it did not yet arrive at the request location's origin, or it can be observed with $jobs = [PROCESS, NULL, NULL]$ when it arrives at the request's pickup location. This reflects the fact that going from one point in the city to another is not instantaneous.

Central operator decisions. When a decision step is triggered, the central operator can make the following decisions:

- Process unassigned requests. The operator can either:
  - Assign a request to an eligible vehicle (see blue arcs in Figure 4.3);
  - Reject the request;
  - Wait until the next decision step to decide.
- Reposition eligible vehicles (see red arcs in Figure 4.3);

In the next section, we will formulate this problem under the reinforcement learning paradigm, as well as introduce our notations for the next sections.

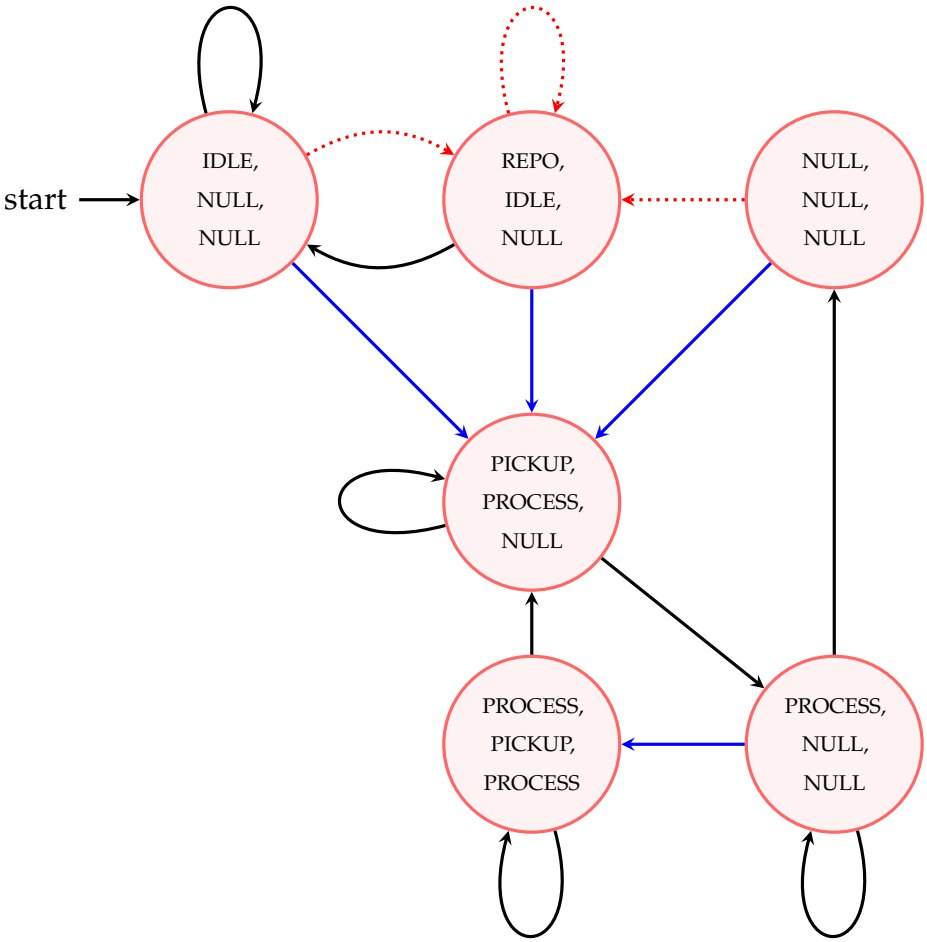


Figure 4.3: Possible changes of vehicles' job types over the time steps. Blue arcs indicate request assignment conditions, which are subject to the time limit constraint. Red arcs represent repositioning conditions.

4.3 Reinforcement learning formulation

To formulate this problem as a reinforcement learning problem, we identify each component of the reinforcement learning framework: the environment, agent, state, actions, reward.

Agent. It is responsible for making decisions (actions) at each decision step, given the current state of the environment. In the problem description, this is what we called the central operator. It handles the fleet of vehicles and assigns them new tasks. It also decides whether a request should be treated, delayed or rejected.

Environment. The city of Manhattan constitutes the environment of this problem. Its main characteristics are the road network structure, the idling locations, the division into zones, the traffic congestion, the stochastic travel speeds between zones and the requests. Moreover, the Manhattan distance is used to compute all the distances. Changes in the environment during the interactions with the agent can be observed through the states' components, as described below.

State. At each decision step t, the state is represented as a tuple $s_t = (s^{\mathcal{T}}, s^{\mathcal{R}}, s^{\mathcal{V}})$ defining:

- Timing $s^{\mathcal{T}} = (t', d)$, with $t' \in [0, T]$[5] being the time at which the decision step was triggered over a time horizon of $T = 86400$ seconds and $d \in \{0, 1, 2, 3, 4\}$ being the day of the week (0 for Monday), excluding the weekends.
- Pending requests $s^{\mathcal{R}} = (s^{P_t}, s_r^{\mathcal{T}})$. With $P_t = \{0, 1, ..., p_t - 1\}$ the set of p_t pending requests at the time t, $s^{P_t} = \{((o_x^i, o_y^i), (d_x^i, d_y^i)), \forall i \in P_t\}$ being the set of the pending requests' Cartesian coordinates of the origin and destination locations, and $s_r^{\mathcal{T}} = \{r_i^{\mathcal{T}}, \forall i \in P_t\}$ represents the request time at which the requests were first received by the operator.
- Vehicles. $s^{\mathcal{V}}$ defines for each vehicle $v \in \mathcal{V}$:

[5]Note that the decision step t and the environment internal timing t' are different from one another. The step t is reported when an event is triggered, while t' refers to the time elapsed since midnight.

- (x_v, y_v) the Cartesian coordinates of its location.
- $[job_i(v) \in \{IDLE, REPO, PICKUP, PROCESS, NULL\}, i \in \{1,2,3\}]$ represents the job types. There are 3 jobs per vehicle.
- $(jo_v^i, jd_v^i),\ i \in \{1,2,3\}$ represents the job origin and destination Cartesian coordinates, i.e., $jo_v^i = (xo_v^i, yo_v^i)$ and $jd_v^i = (xd_v^i, yd_v^i)$, when relevant $(job_i(v) \neq NULL,\ i \in \{1,2,3\})$

We consider that the vehicles share the same environment's state. This means that the agent makes decisions for all the vehicles regarding this current state configuration. This might not be fully accurate since in practice the new state would have to be recalculated each time the agent decides for a given vehicle. This approach reduces the complexity of calculating individual states for each vehicle separately, which can be computationally intensive, especially in environments with numerous vehicles.

Action space. At each decision step t, the operator specifies for each vehicle and request:

- req_rejections $\in \{0,1\}^{p_t}$ a vector of length p_t that indicates whether a request is rejected (1) or not (0), i.e., req_rejections$_i = 1$ implies that $request_i$ is rejected.
- req_assgts $\in \{\mathcal{V} \cup \{V\}\}^{p_t}$ a vector defining the vehicle assigned to each request. A request may be set to *wait* (value V).
- reposition $\in \{L \cup \{|L|\}\}^V$ represents where the vehicles will be repositioned, with L the set of lots (predefined idling locations grouped into zones and represented by their Cartesian coordinates $\{(x_l^i, y_l^i), \forall i \in L\}$). Vehicles can be assigned to any idling location $l \in L$ or not repositioned (value $|L|$).

Thus, an action a_t is defined as a tuple of (req_rejections, req_assgts, reposition)[6].

We should observe that as the number of pending requests p_t changes over time, the req_rejections and req_assgts vectors have a variable length (possibly zero length vectors).

[6]The environment manages inconsistencies, for example if rejected requests are assigned to vehicles, then they are ignored.

Initialization. The initial state s_0 begins at $t' = 0$, the day of the week d is initialized randomly, with no pending requests. The vehicles are idling at a lot, with the $jobs$ queue being $jobs(v) = [IDLE, NULL, NULL]$, $\forall v \in \mathcal{V}$.

Reward. We consider a shared reward $r_t = R(s_t, a_t)$ among the vehicles. A shared reward will encourage cooperation between the vehicles [102]. The reward function $R(\cdot, \cdot)$ is defined as the total sum of the incomes earned by all the vehicles minus the travel costs, as defined in Equation 4.1.

4.4 Solution methods

In this section, we describe the four different solution methods we designed to tackle the problem. The first one relies on completely learning Q-values for repositioning and matching; the second one is a hybrid policy with learned Q-values for matching and a handcrafted nearest neighbor policy for repositioning: the third one is in between the first two policies, it learns the Q-values for matching, and Q-values for selecting the repositioning zone, combined with a nearest neighbor strategy to determine the repositioning lot. The last method is a fully handcrafted heuristic based on a nearest neighbor strategy. Furthermore, we have decided not to reject any customer requests, for all the strategies. Regarding, request's wait, we decided that a customer will wait if and only if all vehicles are busy serving other customers.

Our learning-based agents manage both request assignments (matching) and vehicle repositioning. We tackle the problem from the vehicles' perspective, i.e., at a given state s_t, the action a_t corresponds to a set of actions for each vehicle ($a_t = \{a_t^v, \forall v \in \mathcal{V}\}$). From the vehicle's perspective, an action corresponds to either request assignment or repositioning. This formulation of the action space reduces its size compared with the original formulation in section 4.3.

We train the agents using an n-step Dueling Deep Q-learning (D2QN) algorithm to estimate the Q-values [189]. In D2QN, we estimate both the value of a state $V(s_t)$ and the advantage of each state-action pair $A(s_t, a_t)$. The Q-value of a state-action pair is therefore computed as following: $Q(s_t, a_t) = V(s_t) + A(s_t, a_t)$.

The main advantage of this decomposition is to separate between the common estimation (the value of the state) from the differentiating estimation (the advantage of taking an action on given a state), which often leads to a more accurate Q-value estimation [189]. For this, we use two different neural networks for each quantity to estimate. The first one is the value of a state $V(s_t)$, and the second one is the relative advantage of choosing an action in that state $A(s_t, a_t^v)$, $\forall v \in \mathcal{V}$. Moreover, in the final estimation of the Q-values, the mean advantage of each state-action pair is subtracted from the computed advantage to increase the stability of the optimization [189]. Therefore, for each vehicle $v \in \mathcal{V}$:

$$Q(s_t, a_t^v) = V(s) + \left(A(s_t, a_t^v) - \frac{1}{|\mathcal{A}^v_{DQN}|} \sum_{a_t'^v \in \mathcal{A}^v_{DQN}} A(s_t, a_t'^v)\right)$$

where $\mathcal{A}^v_{DQN}$ is the set of all possible actions that the vehicle can take using the D2QN neural network. This action set is detailed later on, since it differs from one solution strategy to another.

In addition, the D2QN algorithm considers two different estimators for the Q-values: a Q-network and a Target-network. The Q-network represents our learning agents. It estimates the expected cumulative rewards that each vehicle would earn by taking an action. The Target network is a copy of the Q-network, but its parameters are fixed for a certain number of iterations. It is periodically updated using the Q-network parameters to help track the changes in the optimal policy over time, during the learning phase.

During the interactions with the environment, we store the agent's experiences as a tuple (s_t, r_t, a_t, s_{t+1}), at each decision step t, in a replay memory buffer. This replay memory will be used later on to sample experiences to train our Q-network. This memory of size M is cyclical; older experiences are overwritten by newer ones when it is full. In the standard Deep Q-learning approaches, including D2QN, this memory is updated with a new experience at each step. In the case of our problem, delayed rewards issues may occur because the vehicles need a certain time to process a request, and thus to receive a positive reward. The use of n-step Q-learning [171] allows the model to wait for n decision steps before receiving the final reward; thus, the future reward estimation is more accurate. We store

the transitions in our replay memory with n-step delay $(s_t, r_{t:t+n}, a_t, s_{t+n})$ where $r_{t:t+n} = \sum_{i=0}^{n} \gamma^i r_{t+i}$ and γ being a discount factor (usually a value close to 1).

To ensure exploration, we use an $\epsilon-$greedy strategy. We take a random action with a probability ϵ, otherwise, we act according to our Q-network policy. At the beginning of training, we favor exploration by random actions, but as the training progresses, they will become less frequent. To do so, we use an exponential decay strategy for epsilon [124], which is defined as: $\epsilon = \epsilon_f + (\epsilon_i - \epsilon_f) \cdot exp\left(\frac{-total\_steps}{\epsilon_d}\right)$, where $\epsilon_i, \epsilon_f, \epsilon_d$ are the initial, final, and decay step values, and $total\_steps$ counts the number of steps since the beginning of the learning process. At the beginning $\epsilon = \epsilon_i$, then it converges towards ϵ_f when $total\_steps$ gets bigger.

The loss function during training is defined across all vehicles using of batch of B experiences sampled from the replay buffer (see Equation 4.2). It is computed using the mean squared error between:

- target value y_i^v defined as the sum of the n-step delayed reward $r_{t:t+n}^{(i)}$ and the maximum Q-value at state s_{t+n} estimated by the Target network Q_{θ^T} for each vehicle,
- the Q-value at state s_t estimated by the Q-network for each vehicle.

$$L(\theta) = \frac{1}{B \times V} \sum_{i=1}^{B} \sum_{v \in \mathcal{V}} (y_i^v - Q_\theta(s_t^{(i)}, a_t^{(i),v}))^2 \tag{4.2}$$

with $y_i^v = r_{t:t+n}^{(i)} + \max_{a^v \in \mathcal{A}_{DQN}^v} Q_{\theta^T}(s_{t+n}^{(i)}, a^v)$.

The Algorithm 7 gives an overview of the training loop of our deep neural network model. It is the same for all our Deep Q-networks.

In what follows, we describe the main differences between the Deep Q-networks strategies (DQN).

DQN_ALL_LOTS. This Q-network outputs Q-values per request and per vehicle. For each vehicle, our model outputs a Q-value for each pending request $Qreq_i, \forall i \in P_t$, and a Q-value per reposition lot $Qrep$. When there is no pending request, we compute a dummy $Qreq$ value by using a vector of zeros as a representation of a request. Therefore, at each decision step, we have $m = \max(p_t, 1)$

Algorithm 7: The Q-learning algorithm used to train the DQN

Inputs: env: ride-hailing environment, DQN parameters θ, number of episodes E, batch size B, memory size M, target network update frequency M_{update}, DQN learning frequency M_{freq}, initial epsilon ϵ_i, final epsilon e_f, epsilon decay ϵ_d, n-step n, discount factor γ, learning rate lr

$\theta^T \leftarrow \theta$ // initialize target network parameters

$\epsilon \leftarrow \epsilon_i$

$\mathcal{M} \leftarrow []$ // initialize empty memory

$total\_steps \leftarrow 1$

for $e \leftarrow 1$ *to* E **do** // train for E steps

  $s_0 \leftarrow$ env.reset() // get initial observation

  for $t \leftarrow 0$ *to* $T-1$ **do** // perform T steps until episode terminaison

    $(x^{\mathcal{T}}, x^{\mathcal{R}}, x^{\mathcal{V}})_t \leftarrow$ getFeatures(s_t)

    if *random()* $< \epsilon$ **then**

      $a_t \leftarrow$ env.get_random_action()

    else

      $a_t \leftarrow$ get_action_from_dqn($Q_\theta, (x^{\mathcal{T}}, x^{\mathcal{R}}, x^{\mathcal{V}})_t$)

    end

    $s_{t+1}, r_t \leftarrow env.step(a_t)$

    store_n_step_transition($\mathcal{M}, M, (s_t, r_t, a_t, s_{t+1}), n, \gamma$)

    if $total\_steps \mod M_{freq} = 0$ **then**

      $\mathcal{B} \leftarrow$ sample($\mathcal{M}, B$) // uniformly sample a batch of transitions from memory

      learn($Q_\theta, Q_{\theta^T}, \mathcal{B}, lr$) // computes the loss as defined in Equation 4.2 and performs gradient descent using Adam optimizer on Q_θ.

    end

    if $total\_steps \mod M_{update} = 0$ **then**

      $\theta^T \leftarrow \theta$

    end

    $\epsilon \leftarrow \epsilon_f + (\epsilon_i - \epsilon_f) \cdot exp\left(\frac{-total\_steps}{\epsilon_d}\right)$

    $total\_steps \leftarrow total\_steps + 1$

  end

end

$Qreq$ values to compute. In the environment, there are 302 lots spread across the city. Thus, for each vehicle $v \in \mathcal{V}$, its associated Q-values are as follows: $Q^v_{DQN} = \{Qreq^v_1, ..., Qreq^v_m, Qrep^v_1, ..., Qrep^v_{302}\}$. So at each decision step, our DQN computes $V \times (m + 302)$ values. The associated action set per vehicle can be represented as: $\mathcal{A}^v_{DQN} = \{serve^v_1, ..., serve^v_m, rep^v_1, ..., rep^v_{302}\}$. The advantage of this representation is that it considers all the problem aspects in detail, especially for the repositioning part. All the lots have an associated Q-value. The main drawback is that the action space is huge, and can become bigger as the number of vehicles and pending requests grow. Let us note that in our case, we separate the estimation of the $Qreq$ and the $Qrep$ values by using two separate subnetworks for each, since we will use different features to estimate each Q-value type (more details in our neural network architecture in section 4.5). Moreover, we do not explicitely handle the case when the vehicle does not reposition. We argue that

the Q-network can choose in the next decision step to reposition the vehicle to the same idling location where it is at the current decision step, which would be equivalent to instructing it to wait at its idling location.

DQN_NN hybrid strategy. It estimates a Q-value for each pending request, and a dummy Q-value when no pending request is available. For repositioning, it considers all the available lots, but uses a handcrafted nearest neighbor strategy; each available vehicle is assigned to the nearest lot in terms of Manhattan distance. Since we used two different subnetworks for request assignment and vehicle repositioning in the DQN_ALL_LOTS strategy, we decided to take directly the resulting request assignment subnetwork, without retraining it. Our choice is motivated by two research questions: (1) is the repositioning subnetwork's policy a nearest neighbor? (2) otherwise, is the repositioning subnetwork's policy better or worse than a nearest neighbor?

DQN_ZONES. This strategy is in between the previous two strategies. For each vehicle, our model outputs a Q-value for each pending request $Qreq_i, \forall i \in P_t$, another Q-value for the null action that indicates to the vehicle to not reposition Q_{NULL} and a Q-value per reposition zone $Qrep$. When there are no pending requests, we compute a dummy $Qreq$ value. Since the lots are grouped into zones, our Q-network selects the zone where to reposition instead of directly determining the lot where to reposition. Then, we use a nearest neighbor strategy, in terms of Manhattan distance, to select the lot in the chosen zone. Computing a Q-value per zone reduces the cardinality of the output of our DQN, since there are only 50 zones (out of the 61 available zones) where the lots are situated. For each vehicle $v \in \mathcal{V}$, its associated Q-values are as follows: $Q^v_{DQN} = \{Qreq^v_1, ..., Qreq^v_m, Q^v_{NULL}, Qrep^v_1, ..., Qrep^v_{50}\}$. So at each decision step, our DQN computes $V \times (m + 1 + 50)$ values. The associated action set per vehicle is therefore: $\mathcal{A}^v_{DQN} = \{serve^v_1, ..., serve^v_m, NULL, rep^v_1, ..., rep^v_{50}\}$.

Nearest Neighbor. This is a fully handcrafted strategy, where vehicles' assignments and reposition are determined by computing the Manhattan distances between each vehicle and pickup location of a request and each vehicle and lot. The

nearest point is then selected as the destination of the vehicle. We will use this strategy as a baseline to compare the performance of our learning-based and hybrid policies.

Table 4.3 summarizes the solutions methods used to tackle the problem. DQN_ALL_LOTS, DQN_NN, DQN_ZONES agents use a Q-value estimation for matching vehicles to requests. At inference, when tackling instances unseen during training, the vehicle with the maximum Q-value is chosen to handle the request. DQN_ALL_LOTS agent chooses the lot with the maximum Q-value for repositioning, while DQN_NN agent uses a nearest neighbor strategy. DQN_ZONES agent chooses the zone with the maximum Q-value for repositioning, and assigns the vehicle to the nearest lot in that zone.

strategy	matching	repositioning
DQN_ALL_LOTS	Q-value	Q-value at lot level
DQN_NN	Q-value	Nearest neighbor (NN)
DQN_ZONES	Q-value	Q-value at zone level and NN for lot selection
Nearest Neighbor	NN	NN

Table 4.3: Summary of the solution methods.

4.5 The deep neural network architecture

In this section, we provide a description of the deep neural network we defined to tackle the problem. Contrary to other works [102, 77], our main focus is to design a deep neural network that can tackle a variable number of requests with a variable number of vehicles. Thus, we do not have to train a single model each time the number of requests or the number of vehicles changes in the environment. *In the following,* $\mathbf{W}$ *and* $\mathbf{b}$ *denote the weights and biases of our DQN layers.*

Decision step features. We define the decision step features to feed to our deep Q-network from the environment's state $s_t = (s^{\mathcal{T}}, s^{\mathcal{R}}, s^{\mathcal{V}})_t$ as follows:

- $\mathbf{x}^{\mathcal{T}} \in \mathbb{R}^6$ is the time features vector, it is made of the concatenation of relative time t/T and a one hot encoding of the 5 days of the week excluding

the weekend (dow). We use a one hot encoding representation not to imply an order relationship between the days of the week.

- $\mathbf{x}_r^{\mathcal{R}} \in \mathbb{R}^5,\ \forall r \in P_t$ is the request feature vector. Each request is represented using a concatenation of:
  - its relative triggering time t_r/T,
  - its origin's normalized coordinates[7],
  - its destination's normalized coordinates.

  If there is no pending request, $\mathbf{x}_r = 0_{\mathbb{R}^5}$.
- $\mathbf{x}_v^{\mathcal{V}} \in \mathbb{R}^{14},\ \forall v \in \mathcal{V}$ is the vehicle features vector. It is represented using the concatenation of the normalized vehicle's current location coordinates ($2D$) and the normalized vehicle's $jobs$ origin and destination coordinates ($3 \times 2 \times 2D$).

Initial embeddings. We first compute time ($\mathbf{e}^{\mathcal{T}}$), requests' ($\mathbf{e}^{\mathcal{R}}$), and vehicles' embeddings ($\mathbf{e}^{\mathcal{V}}$) adopting the same approach:

$$\mathbf{e}_{(\cdot)}^{u} = ReLU(\mathbf{W}_u\ \mathbf{x}_{(\cdot)}^{u} + \mathbf{b}_u), u \in \{\mathcal{T}, \mathcal{R}, \mathcal{V}\}$$

with $\mathbf{W}_u \in \mathbb{R}^{d \times d_u}$ and $\mathbf{b}_u \in \mathbb{R}^d$.
These embeddings are a first mapping of each of the state features into a d dimensional vector space. They offer an initial alternative representation of each feature, but these representations are independent of each other, since different parameters are used for each type. For example, the vehicles are not aware of the requests' or time representations.

Vehicles embeddings. To enhance the representation of the vehicles, we define an attention mechanism. For this, we use the same formalism as used by the Transformer architecture by defining queries, keys, and values [178]. The queries

[7]The coordinates (x, y) are bounded in ranges $[x_{\min}, x_{\max}]$ and $[y_{\min}, y_{\max}]$, respectively. These ranges are predefined in the environment. We use Min-Max Normalization on the coordinates to make the learning process more stable and to speed up the convergence of the deep Q-network.

will confront the keys to compute attention scores that take into consideration the requests representations. The resulting scores will then be used to compute a new vehicle representation considering other vehicles' representations. We use a scaled dot-product attention that is faster to compute. It is used to get a new representation of each vehicle that considers the requests and the other vehicles representations.

We define the queries by a linear transformation of the concatenation of each vehicle embedding with the request embedding followed by a $ReLU$ activation function:

$$\mathbf{q}_{rv} = ReLU(\mathbf{W}_q \ [\mathbf{e}_v^{\mathcal{V}}; \mathbf{e}_r^{\mathcal{R}}] + \mathbf{b}_q), \ \forall r \in P_t, v \in \mathcal{V}$$

with $\mathbf{W}_q \in \mathbb{R}^{d \times 2d}$ and $\mathbf{b}_q \in \mathbb{R}^d$.

The keys are defined by another linear transformation of the vehicle embeddings:

$$\mathbf{k}_v = \mathbf{W}_k \ \mathbf{e}_v^{\mathcal{V}} + \mathbf{b}_k$$

with $\mathbf{W}_k \in \mathbb{R}^{d \times d}$ and $\mathbf{b}_k \in \mathbb{R}^d$.

We consider the vehicle embeddings $\mathbf{e}_v^{\mathcal{V}}, \forall v \in \mathcal{V}$ without applying any transformation for the values.

For each request $r \in P_t$ and for each pair of vehicles $(i, j) \in \mathcal{V} \times \mathcal{V}$, we get an attention score using a scaled dot-product followed by the softmax function, i.e.,

$$\alpha_{rij} = softmax\left(\frac{\mathbf{q}_{ri} \cdot \mathbf{k}_j^\top}{\sqrt{d}}\right)$$

with $0 < \alpha_{rij} \leq 1$ and $\sum_{j \in \mathcal{V}} \alpha_{rij} = 1$.

Finally, we get a vehicle representation per request as a convex sum of all other vehicles embeddings, where the convex coefficients are the attention coefficients, i.e.

$$\mathbf{h}_{rv} = \sum_{j \in \mathcal{V}} \alpha_{rvj} \mathbf{e}_j^{\mathcal{V}}$$

The value network. The value network is a multi-layer perceptron with 2 layers and a $ReLU$ activation function in between for non-linearity. It takes as input a concatenation of:

1. a global fleet embedding defined using a global average pooling of all vehicles embeddings $\bar{\mathbf{h}} = \frac{1}{p_t \times V} \sum_{r \in P_t} \sum_{v \in \mathcal{V}} \mathbf{h}_{rv}$,
2. a global request embedding also defined using a global average pooling of all pending requests $\bar{\mathbf{e}}_{\mathcal{R}} = \frac{1}{p_t} \sum_{r \in P_t} \mathbf{e}_r^{\mathcal{R}}$,
3. the time embedding $\mathbf{e}^{\mathcal{T}}$

$$V(s) = \mathbf{W}_{s_1} \; ReLU(\mathbf{W}_{s_2} \; [\bar{\mathbf{h}}; \bar{\mathbf{e}}_{\mathcal{R}}; \mathbf{e}^{\mathcal{T}}] + \mathbf{b}_{s_2}) + \mathbf{b}_{s_1}$$

with $\mathbf{W}_{s_1} \in \mathbb{R}^{1 \times d}, \mathbf{b}_{s_1} \in \mathbb{R}, \mathbf{W}_{s_2} \in \mathbb{R}^{d \times 3d}, \mathbf{b}_{s_2} \in \mathbb{R}^d$.

The advantage network. Since we have a variable number of pending requests, we separate the computation of the requests advantages and the repositioning advantages using two different neural networks.

The request advantage subnetwork uses a scaled dot-product attention between vehicles representations $\mathbf{h}_{rv}$ and a query $\mathbf{q}_r^{\mathcal{RT}}$.

$$adv_{rv} = \frac{\mathbf{q}_r^{\mathcal{RT}} \cdot \mathbf{h}_v^\top}{\sqrt{d}}, \quad \forall r \in P_t, \forall v \in \mathcal{V}$$

The query is formed using a linear transformation of the concatenation of each request embedding with the time embedding, followed by a $ReLU$, i.e.

$$\mathbf{q}_r^{\mathcal{RT}} = ReLU(\mathbf{W}_{\mathcal{RT}} \; [\mathbf{e}_r^{\mathcal{R}}; \mathbf{e}^{\mathcal{T}}] + \mathbf{b}_{\mathcal{RT}}), \; \forall r \in P_t$$

with $\mathbf{W}_{\mathcal{RT}} \in \mathbb{R}^{d \times 2d}$ and $\mathbf{b}_{\mathcal{RT}} \in \mathbb{R}^d$.

The repositioning advantage subnetwork is a multi-layer perceptron with 2 fully connected layers and a $ReLU$ activation function in between. For the DQN_ZONES approach, it outputs one advantage score per zone where vehicles can reposition in addition to a score for the no reposition action, while for the DQN_ALL_LOTS policy, it outputs one score per lot. The repositioning advantage subnetwork takes as input a representation of vehicles embedding computed as a mean aggregation of vehicles embeddings over the requests ($\bar{\mathbf{h}}_v = \frac{1}{p_t} \sum_{r \in P_t} \mathbf{h}_{rv}, \quad \forall v \in \mathcal{V}$) concatenated

with a global time request representation $\mathcal{G}$ computed as follows:

$$\mathcal{G} = ReLU(\mathbf{W}_g \ [\bar{\mathbf{e}}_{\mathcal{R}}; \mathbf{e}^{\mathcal{T}}] + \mathbf{b}_g)$$

with $\mathbf{W}_g \in \mathbb{R}^{d \times 2d}$ and $\mathbf{b}_g \in \mathbb{R}^d$.

4.6 Computational results

In this section, we present the computational results of the various strategies we designed. We begin by reporting the results during the training phase for our learning-based agents. We, then, confront these agents to the remaining solution strategies on unseen instances, to evaluate the generalization capabilities of our proposed approaches. Regarding the environment reward and episode length, the time horizon is $T = 24$ hours, the per kilometer reward is $C_D = 4.02\$/km$, the fixed reward is $C_F = 10.75\$$, $TC = 0.53\$/km$, $w^* = 5$ minutes. For the environment, we used pyhailing[8] which uses the same programming interfaces as OpenAI gym.

4.6.1 Model training

We first train two different agents. The first one learns to assign vehicles to requests and to reposition (or not) a vehicle to a zone (DQN_ZONES), while the second one (DQN_ALL_LOTS) learns to assign vehicles to requests and to assign a vehicle to a lot. The first model took approximately 12 hours to train on GTX 1660 super GPU, while the second one, took a whole day for training. We trained each model over 300 episodes, with a fleet of 14 vehicles and 1400 requests per day. Table 4.4 displays the hyperparameters used during the training phase. The reported values are the result of a manual tuning process aimed at achieving the best possible results for our model.

Figure 4.4 represents the evolution of total reward per training episode for the DQN_ZONES (in orange) and DQN_ALL_LOTS (in blue) policies. As we can observe, both models can learn a policy, improving at each training episode. In the first forty (40) episodes, we can observe that the two policies seem to have

[8]available at: https://pypi.org/project/pyhailing/

Hyperparameter	Value
Number of episodes E	300
MAX_TIME	60 seconds
Training seed	321
Discount factor γ	0.999
Replay memory size M	100000
Training frequency M_{freq}	100
Batch size B	32
Embedding dimension d	64
Target network update frequency M_{update}	5000
Initial epsilon ϵ_i	1
Final epsilon ϵ_f	0.1
Epsilon decay steps ϵ_d	75000
n-step n	3
learning rate lr	10^{-3}

Table 4.4: Hyperparameters used during training.

slightly similar performances, this may be due to the use of the ϵ-greedy policy, which favors random actions for the first episodes. From the episode 40 to 100, the DQN_ALL_LOTS policy continues to change towards a better policy, in terms of total reward per episode. From episode 100 onwards, the policy stabilizes at around 6000$ reward per episode. On the other hand, the DQN_ZONES model continues its exploration of new policies from episode 40 to 200, where it begins to stabilize at around 13000$ reward per episode. As we can see, the DQN_ZONES final policy outperforms the DQN_ALL_LOTS final policy. One possible explanation is that the action space of the DQN_ZONES model is smaller than the one of DQN_ALL_LOTS, thus it converged towards a better local optimum.

4.6.2 Evaluation

We evaluate our models on two different instance sizes: the small instances have similar size as the ones used during training, 14 vehicles and 1400 requests per episode, while the big instances consider a fleet of 100 vehicles and 10000 requests per episode. We use a different seed than the one used for training (20151101).

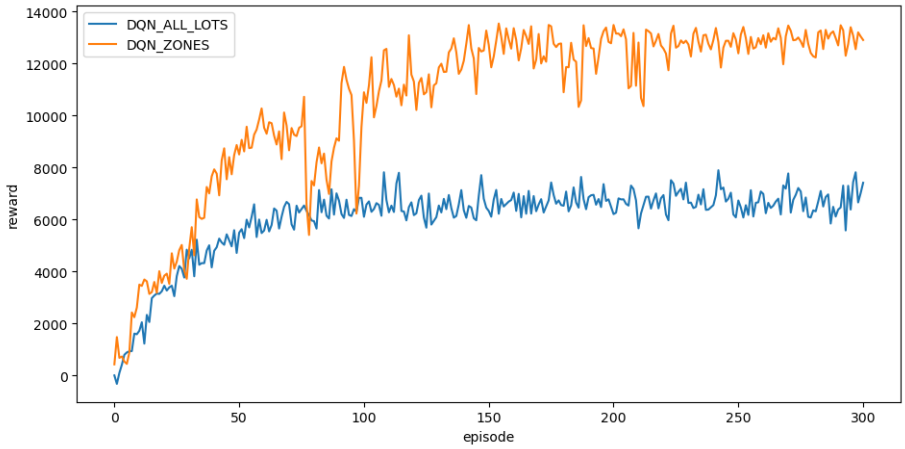


Figure 4.4: Evolution of total reward per episode during training for DQN_ZONES and DQN_ALL_LOTS models.

Moreover, the evaluation is made over 50 episodes, for both sizes. In addition to the mentioned strategies, we add the random policy as a trivial baseline[9].

Figure 4.5 and Table 4.5 summarize the results of all the strategies on the small instances, while Figure 4.6 and Table 4.6 do the same on the big instances. We observe two different behaviors in both cases, that we will describe hereafter.

Figure 4.5 shows a box plot that summarizes the evolution of the total reward per episode between our designed strategies over the small size instances. As we can see, all the models outperforms the random policy, and DQN_ZONES outperforms DQN_ALL_LOTS which confirms the results observed during training. Surprisingly, the total rewards of DQN_ZONES during evaluation decreased compared with the ones achieved during training, but we observe the inverse for the DQN_ALL_LOTS policy, which achieves better total rewards than what we observed during training. This may be due to a sensitivity to the seed used during the evaluation. On another front, we observe that the nearest neighbor (NN) policy has similar performances to DQN_ZONES, but slightly outperforms it. Finally, when changing the repositioning policy to a nearest neighbor policy (DQN_NN), we observe that we obtain the best model with a median near 12000$. This contributes to answering our questions on the DQN_ALL_LOTS model. These observations

[9]This strategy chooses randomly which vehicle to match with a request and where vehicles reposition.

suggest that the learned repositioning policy is different from the nearest neighbor policy, and that using the latter produces better results. Table 4.5 presents more details on the results on the small size instances. The first four columns confirm the observations in the figure. The last column offers additional information on the average percentage of fulfilled requests per episode. We can observe that none of the policies can satisfy more than 40% of the requests. One possible explanation is that the fleet size is too small: they are most of the time unable to arrive within the time limit of $w^* = 5$ minutes. Another reason is that the repositioning strategies are far from optimal. Thus, there is still room for improvement on the small size instances.

	Small instances				
	mean	std	min	max	avg. served requests (%) (±std)
RANDOM	796.66	306.13	154.50	1738.54	15.67 (± 0.90%)
DQN_ZONES	10386.41	684.12	**9250.16**	12049.81	36.55 (± 2.29%)
DQN_ALL_LOTS	8285.19	596.11	6474.06	9740.53	29.57 (± 2.02%)
DQN_NN	**11895.58**	1011.10	9228.01	**13519.65**	**40.61** (± 3.70%)
Nearest Neighbor (NN)	10575.70	972.09	8304.09	12332.02	36.48 (± 3.61%)

Table 4.5: Summary of the evaluation results on 50 small size instances, reporting the mean, standard deviation, minimum, maximum achieved rewards, and the average number of fulfilled requests.

Figure 4.6 shows a box plot of the evolution of the total reward per episode between our policies for the big size instances. For our learning-based policies, we can observe that they are all able to generalize well to instances with bigger size than the ones seen during training. We observe that DQN_ZONES outperforms the other models, and has better generalization properties than the others. It is followed by the Nearest Neighbor (NN) policy. Hybridizing DQN and nearest neighbor for repositioning (DQN_NN) still brings improvement compared with a policy that fully learns to take both decisions, but this time, we observe only a slight improvement, compared with what we observed on the small size instances. This still confirms that for DQN_ALL_LOTS, it is difficult to learn a good repositioning policy, and that a nearest neighbor repositioning improves the total rewards. This result is emphasized by Table 4.6 which shows that DQN_ALL_LOTS and DQN_NN serve the same number of requests (on average), but DQN_NN achieves

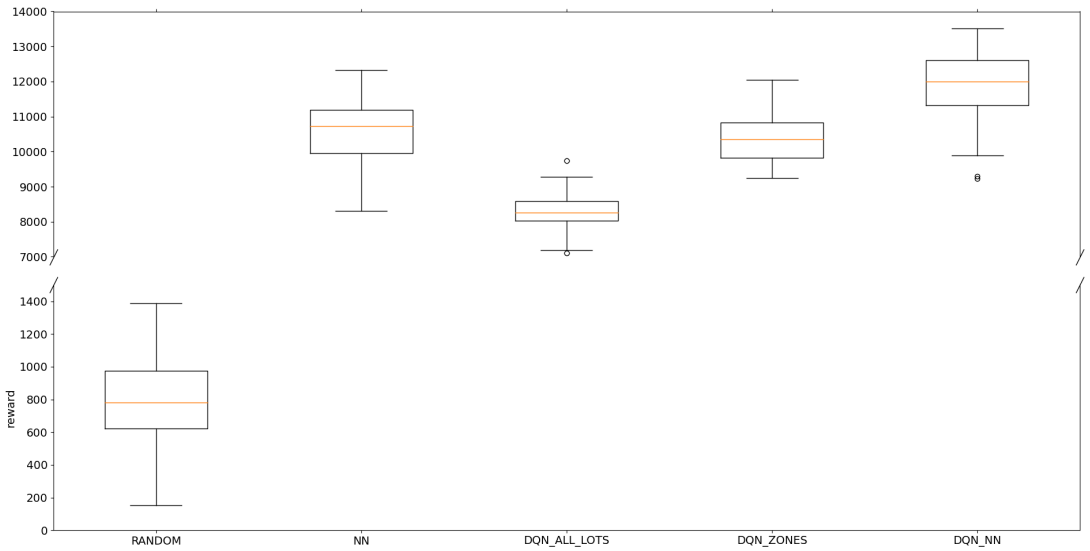


Figure 4.5: Box plot summarizing the evolution of the total reward per episode for the different resolution strategies over 50 episodes, using instances with 14 vehicles and 1400 requests per day.

better total rewards. This time, all the models are able to serve more than 50% of the requests, with the maximum being for DQN_ZONES with 67.27% of the requests that are being fulfilled. Changes in the observed patterns may be due to the large size of the fleet of vehicles. Furthermore, a hybrid repositioning with a deep neural network that determines the zones and a nearest neighbor strategy that chooses the lot seems to give better results. In the worst case (small size instances), it achieves results similar to the handcrafted policy (NN).

	Big instances				
	mean	std	min	max	avg. served requests (%) (±std)
RANDOM	7440.66	935.92	4988.77	9459.72	16.19 (± 0.39%)
DQN_ZONES	**139420.00**	7235.41	96741.11	**145487.01**	**67.27** (± 3.5%)
DQN_ALL_LOTS	121424.29	2056.10	115926.93	125988.02	58.22 (± 1.06%)
DQN_NN	123098.96	2946.82	117092.79	129051.64	58.22 (± 1.44%)
Nearest Neighbor (NN)	132086.59	4800.74	**118904.99**	141314.62	62.98 (± 2.35%)

Table 4.6: Summary of the evaluation results on 50 big size instances, reporting the mean, standard deviation, minimum, maximum achieved rewards, and the average number of fulfilled requests.

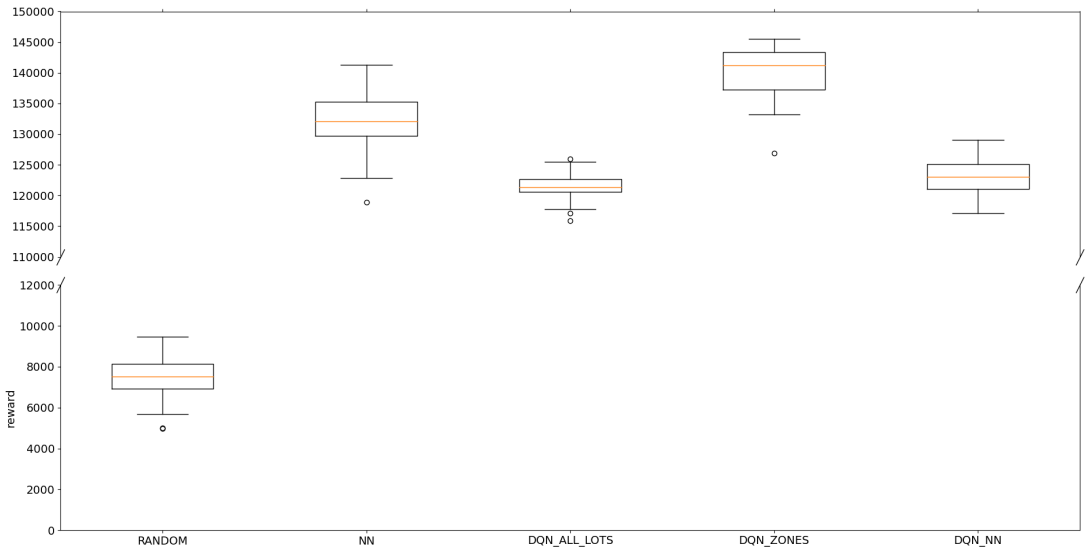


Figure 4.6: Box plot summarizing the evolution of the total reward per episode for the different resolution strategies over 50 episodes, using instances with 100 vehicles and 10000 requests per day.

4.7 Conclusion and perspectives

In this chapter, we investigated a use case of deep Q-learning on the Ride-hailing problem, a dynamic and stochastic variant of the VRP. For this, we formulated the problem under the reinforcement learning framework. We then, designed three different deep learning-based policies for tackling the problem: the first one relies on completely learning Q-values for repositioning and request assignment, the second one is a hybrid policy with learned Q-values for the request assignment and a handcrafted nearest neighbor policy for repositioning. The third one, is in between the first two policies: it learns the Q-values for the request assignment, and Q-values for the zones where to reposition; the nearest lot to the vehicle in the chosen zone is then designated as the repositioning lot. Our designed deep neural network for this problem can handle an arbitrary number of requests with an arbitrary number of vehicles. This means that we can train our neural network on instance sizes, and deploy it to be used on instances of different sizes. We experimented on small size instances and big size instances, where we observed

good generalization properties of the policies. We observed better results for the hybrid policies than the leaning-based policy or the handcrafted policy. This suggests to us that we should investigate more on hybridization between learning-based models and handcrafted heuristics.

On the other hand, future works should investigate more on the deep neural network architectures to use. While our deep neural network has a desirable property of being able to tackle an arbitrary number of requests with an arbitrary number of vehicles, it may not be complex enough to grasp all the important features. This can be one plausible hypothesis to explain that the full learning-based policy does not perform better than the hybrid models. One possible candidate model for this problem are temporal graph neural networks [156, 52] which can make a representation of dynamic graphs. Finally, another interesting line of research would be the investigation of model-based deep reinforcement learning [139]. In this thesis, we focused only on model-free deep reinforcement learning, but it may be interesting to learn a model of the environment, including the dynamics and the stochastic nature of the requests.

Conclusion and future work

The intersection between operations research and artificial intelligence has the potential to produce considerable benefits, yet it remains a relatively unexplored field. The work initiated in this thesis is part of the efforts made to bring the two disciplines closer together by studying the application of machine learning tools on combinatorial optimization problems, exploiting recent developments in deep learning and reinforcement learning. Through this manuscript, we propose three contributions to a family of combinatorial optimization problems widely studied in the literature, namely vehicle routing problems. Specifically, we consider the Capacitated Vehicle Routing Problem (CVRP), the most basic variant, which is not less interesting for our study due to its complexity ($\mathcal{NP}$-hard). This problem is widely studied in the literature, which allows us, by comparing ourselves to known results, to objectively assess the performances of our contributions. The second problem considered is a stochastic ride-hailing problem. Through this problem, we aim to study the performance of neural approaches on stochastic variants of the vehicle routing problem.

First, we review the current state of knowledge on vehicle routing problems, deep neural networks, and reinforcement learning. We contextualize vehicle routing problems, their existing variants, and the various algorithms developed over the years. This allows us to distinguish between static and deterministic problems, where data is known before the start of the resolution and does not involve any form of uncertainty, and dynamic and stochastic problems, where data are revealed throughout the resolution process, and it involves uncertainty.

Secondly, we were interested in understanding the tools that we will mobilize to solve these problems. Deep neural networks are learning models based on the principle of universal approximation. The last decade has seen the emergence and rapid development of new neural network models, architectures, more efficient training algorithms (both in supervised and reinforcement learning), and vast amounts of data. This has allowed the application of these models to a wide range of problems that are considered difficult to solve using explicit resolution approaches, e.g., handcrafted methods.

In the second part of the chapter, we delved into the literature that focuses on the application of these models to tackle vehicle routing problems. We conducted a comprehensive study of over 35 recent contributions. Several interesting conclusions emerge for us: deep neural network-based models are capable of finding solutions without explicitly specifying how to solve the problem; reinforcement learning is more favored for training neural network models; as with handcrafted heuristics, there are construction and improvement approaches; the quantity of instances needed to train the models is significant (on the order of millions of instances); the majority of the approaches presented are evaluated on small-sized instances; several model exploitation strategies have been implemented to obtain better quality solutions during the evaluation phase.

To conclude the first chapter, we looked at the evaluation protocols of these approaches. Most studies propose to train and evaluate on synthetic data, all coming from the same distribution, to compensate for the lack of data quantity needed for model training. This can pose a problem for a more precise evaluation of performance, since the instances used do not consider the diversity of possible layouts for clients and the depot. To overcome this weakness and better appreciate the performance of these models, we proposed a study on instances of CVRPLib, widely exploited in the handcrafted heuristics' literature. These instances have the advantage of distinguishing several cases of client layouts (random, grouped) and depot layouts (centered, eccentric), as well as a diversity of demand orders and vehicle capacity, and heterogeneous instance sizes.

To achieve this, we reproduced the Attention Model (AM), one of the best performing approaches in the literature, and evaluated it through a single instance search solution method (Active Search algorithm). Our results were able to confirm

the performance orders observed in the literature on instances with a random client layout, but also to show that these models displayed good results on instances with clustered clients. Regarding instance size scale, our evaluations showed that the models provided good quality solutions on small-sized instances, but scaling up is still a significant challenge. In terms of execution time, these were significantly longer than the best heuristic and metaheuristic approaches. However, it is worth noting that our study specifically focused on intensifying the exploration of the search space. Therefore, we diverted these models from their primary use, which is to be trained on a large database of instances and then used for inference on unseen instances during training with exploitation algorithms that can be executed in seconds or minutes.

Chapter two addresses the study of transfer learning in the context of the Neural Combinatorial Optimization framework, through experiments on the Traveling Salesman Problem (TSP) as the source task and the Capacitated Vehicle Routing Problem (CVRP) as the target task, which is a generalization of the former. The study aims to investigate whether a model trained on the source task can be reused to solve the target task. An experimental plan was constructed considering several assumptions: (a) the same model is used for both tasks, (b) relatively few data are available for training on the target task (16k, 32k or 64k), (c) training on the target task is done for a fixed number of epochs. Similarly, different scenarios were considered based on three parameters: (1) the number of data used to train on the source task, (2) the data distribution for the two problems, (3) the instance sizes used for the two problems.

Our various experiments indicate that transfer learning can be beneficial in cases where relatively few data is available for the target task. It improves the asymptotic (final) performance of the model compared to the model without transfer learning while accelerating the learning on few epochs. In addition, models trained on instances of similar sizes and from the same distribution between the source and target tasks yield better results. Finally, in the worst configurations, transfer learning seems not to harm the learning process, given the observed asymptotic performances being at worst comparable to learning from scratch.

Chapter three proposes a novel two-steps algorithm for the CVRP, following the two-steps methods known in heuristic algorithms. This approach combines a deep

neural network with a shortest path algorithm. The deep neural network is used to generate a client visiting order, called a giant tour, which then defines an auxiliary graph on which a shortest path algorithm is applied to obtain the routes and their cost (the Split algorithm). This approach combines the advantages of using a deep neural network without the need to explicitly define the giant tour construction method, with the advantages of Split that extracts the best routes from a giant tour. Moreover, the cost of the routes returned by Split allows us to train the neural network using reinforcement learning. We have designed and evaluated three different neural network models based on Graph Neural Networks (GNNs). In addition, we proposed to introduce the polar angle to the depot as a new raw feature during the encoding phase, which has proven to be particularly useful in improving the convergence of our model towards a better optimum. During evaluation, we found that the GNN-based Transformer yields the best results, and our results are competitive with those of neural network-based constructive methods and heuristic algorithms. Furthermore, our model is faster in training and inference than neural network-based constructive methods. Additionally, our evaluations on CVRPLib confirm the performance of our approach on small and medium-sized instances, regardless of the client distribution.

The final chapter of this thesis is devoted to the study of neural combinatorial optimization approaches on a stochastic and dynamic problem. To this end, we have chosen the ride-hailing problem, which has gained significant interest due to the numerous applications offering this type of service. It represents an excellent case study as it inherently involves dynamic and stochastic aspects. Indeed, it is not possible to know in advance the location of all client requests before the problem-solving process begins. Moreover, there is uncertainty regarding the travel times of vehicles, depending on the starting and ending locations as well as the time of day. Additionally, this problem has two combinatorial aspects: the assignment of vehicles to requests and the repositioning of vehicles to anticipate and to be closer to future requests.

To address this problem, we constructed a deep neural network capable of handling any number of requests while manipulating a variable fleet of vehicles. Additionally, we designed several resolution strategies. In the first strategy, assignment of requests to vehicles, as well as vehicle repositioning, are both managed by

the deep neural network. A second strategy involves managing the assignment of requests to vehicles using deep neural networks and performing repositioning via a nearest neighbor algorithm. Finally, we created a strategy that lies somewhere between the first two. The assignment of requests to vehicles is done using deep neural networks, but for repositioning, the deep neural network predicts the area where the vehicle needs to go, and the idling location is selected using a nearest neighbor algorithm. We trained our models with an n-step Double Deep Q-learning algorithm.On the small size instances, the algorithm that combines a neural network for request assignment and a nearest neighbor approach for repositioning is more effective. On larger instances, the algorithm that combines a zone-based repositioning approach with a nearest neighbor approach is more effective. However, there is still room for improvement in future work to achieve better request satisfaction rates. Our algorithms on average satisfy 40% of requests on small instances and 68% on larger instances. Our approach has the advantage of good generalization properties, as it is capable of handling both instances of the same size as those used during training, and instances of larger sizes, without needing to retrain the model on the new instance size.

Given the state of the art and the work of this thesis, it can be seen that it is possible to develop effective deep neural network-based algorithms to solve combinatorial optimization problems. Although these approaches fell into disfavor at the end of the 1990s in favor of metaheuristics, they have greatly benefited from advances in deep and reinforcement learning, enabling them to produce high-quality solutions. However, there are still challenges to be met by these methods. Firstly, while the results are good, they still fall short of what metaheuristics can achieve. Neural combinatorial optimization is a relatively new field, and many aspects remain to be explored. Therefore, it is possible that in the future, this learning framework applied to optimization problems will become as effective as metaheuristics. In the following, we present several ideas for future research, which we group into five relevant aspects: instances and their nature, representations, algorithmic nature, problems addressed, and explainability.

On instances and their nature: we have observed that most models can handle small and medium-sized instances. Models that are capable of handling larger

instances are hybrids of algorithms known for their efficiency in solving CVRP instances, such as LKH3, HGS-CVRP, and LNS [113, 29]. There is therefore a challenge in ensuring that these models can handle large instances in reasonable computation times. Additionally, the number of instances required for training must also be constrained to account for real-world conditions in the application of these methods. Indeed, it is rare to come across real-world problems where millions of instances are available for training. This will further encourage research to develop more sample-efficient models and training algorithms. Moreover, an aspect that is still neglected is the quality of instances used during training. Future work may include selecting representative instances of the problem to consider the diversity of configurations that may be encountered.

On the representations: In this aspect, we address the question of learning the underlying structure and relationships that link raw data. In Chapter 2, we saw that transfer learning could be a solution when we have two similar problems. This is due to the exploitation of representations made of instances on the source task. Other methods for learning representations can be considered. For example, self-supervised learning [122] could be considered via the definition of relevant pretext tasks for these types of problems. For example, we explored the use of the TSP as a pretext task for learning representations for the CVRP in Chapter 2. Other approaches to self-supervised learning, taking inspiration from image processing, could be considered, such as contrastive learning, which involves differentiating between similar instances (up to a rotation, for example) and different instances [83]. Furthermore, the search for deep neural network architectures suitable for this type of problem can be an interesting research direction, through what has been developed in the field of neural architecture search (NAS) [50]. Although graph neural networks seem to be suitable for routing problems, we have seen in Chapter 3 that the type of encoding (responsible for learning representations) used (GAT, GCN, Transformer) has a significant impact on performance, as does the number of encoding layers used. Therefore, there are chances that a NAS approach can provide us with a better architecture resulting in better results. Finally, raw features can also be subject to further investigation. Although deep neural networks are known to require little human intervention for the explicit

definition of useful raw features, we saw in Chapter 3 that adding the angle to the depot improves neural network performance. We believe that this research direction has not been fully explored and deserves particular attention in future work.

On the algorithmic nature: this concerns both training algorithms and search algorithms used for inference. Regarding training algorithms, while the majority of approaches consider reinforcement learning for its advantage in avoiding the collection of optimal solutions, it is not excluded that supervised learning may lead to better models, especially if coupled with a meticulous selection of instances to build the dataset. Additionally, in the context of reinforcement learning, the reward function admitted in most cases is the total distance traveled. However, nothing excludes that a better reward function may allow for better convergence. Similarly, it is conceivable to use a Reinforcement learning from human feedback (RLHF) approach to solve problems where the objective function is complex to formulate, but where domain experts can qualitatively evaluate the solution based on their experience [34]. Moreover, as we have seen in Chapter 4, hybridizing a learning model with a heuristic can sometimes offer higher quality solutions. This can lead us to rethink what can be delegated to a learning model and what can be delegated to a heuristic [113, 67]. Furthermore, we may consider the solution of a neural network model as an excellent starting solution for a local search. Let us also note that improvements can also be made for the implementations, such as reducing the precision of numerical values of the weights (quantization) or removing unnecessary connections or neurons from the network (pruning). Moreover, better inference times can be obtained with by using more low-level programming languages such as C or C++, especially for the search strategies.

On the problems addressed: from this perspective, we can distinguish the study of complex problems that integrate dimensions that can hardly be captured by classical optimization approaches. This would help us to solve more realistic variants of routing problems. For example, integrating driver experience through a history of their deliveries to better estimate travel times, or predicting customer demand over a time horizon to choose the most appropriate time to serve them. We may

also consider, in this context, models capable of solving a larger family of routing problems, adapting to the presence or absence of certain constraints in the problem. Neural networks are well suited for this type of exercise through the learning of common representations between different variants of the problem.

On the explainability: The black-box nature of neural networks makes it difficult to understand what is actually learned from the data [75]. How can we ensure that what is learned by a neural network corresponds to an algorithm for solving a VRP? Can this algorithm's decisions be explained? What is the relationship between the learned algorithm and the size of the instances it can handle? To understand this, work on explainability of neural networks in the context of combinatorial optimization may be particularly crucial if we want to apply this framework to real-world problems. For example, how can we explain to a driver that they are making a longer tour than another driver? How can we explain that a solution that seems counter-intuitive is better than a more intuitive solution (e.g., nearest neighbor)? How can we explain that neural networks trained on instances of a certain size struggle to generalize to other instance sizes? Furthermore, explainability can help detect biases in the learned algorithms. For example, in the RHP case, we can ensure equitable service by identifying any potential biases that may lead to prioritization or preferential treatment of rich districts over other districts.

Finally, some issues raised in these perspectives are already being researched in a broader framework encompassing deep learning and reinforcement learning. What we emphasize here is the need to adapt them to the framework of neural combinatorial optimization, as presented in the work of this thesis. Furthermore, we highlight that these different perspectives are not mutually exclusive, and it is necessary to consider several aspects as interconnected, such as interpretability, the training algorithm, and the neural network architecture.

Bibliography

[1] L. Accorsi and D. Vigo. A fast and scalable heuristic for the solution of large-scale capacitated vehicle routing problems. *Transportation Science*, 55(4):832–856, 2021.

[2] N. Achuthan, L. Caccetta, and S. Hill. A new subtour elimination constraint for the vehicle routing problem. *European Journal of Operational Research*, 91(3):573–586, 1996.

[3] N. R. Achuthan, L. Caccetta, and S. P. Hill. An improved branch-and-cut algorithm for the capacitated vehicle routing problem. *Transportation Science*, 37(2):153–169, 2003.

[4] C. Ackermann and J. Rieck. A novel repositioning strategy for ride-hailing problems. *12th DIMACS Implementation Challenge: Vehicle Routing Problems*, 2022.

[5] Y. Agarwal, K. Mathur, and H. M. Salkin. A set-partitioning-based exact algorithm for the vehicle routing problem. *Networks*, 19(7):731–749, 1989.

[6] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. The traveling salesman problem. In *The Traveling Salesman Problem*. Princeton university press, 2011.

[7] J. Araque G, G. Kudva, T. L. Morin, J. F. Pekny, et al. A branch-and-cut algorithm for vehicle routing problems. *Annals of Operations Research*, 50(1):37–59, 1994.

[8] M. A. Ardeh, Y. Mei, and M. Zhang. Transfer learning in genetic programming hyper-heuristic for solving uncertain capacitated arc routing problem. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 49–56. IEEE, 2019.

[9] P. Augerat, J. Belenguer, E. Benavent, A. Corberan, D. Naddef, and G. Rinaldi. Computational results with a branch and cut code for the capacitated vehicle routing problem. *Rapport de recherche- IMAG*, 1995.

[10] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *stat*, 1050:21, 2016.

[11] C. Babet, L. Bouvry, M.-F. Brasseur, C. Colussi, M. Jlassi, S. Lambrey, S. Marigot, and C. Rizk. Les chiffres du transport, May 2021. Available at : `https://www.statistiques.developpement-durable.gouv.fr/edition-numerique/chiffres-cles-transport-2021/`, Accessed : 2022-05-03.

[12] Y. Bai, W. Zhao, and C. P. Gomes. Zero training overhead portfolios for learning to solve combinatorial problems. *arXiv:2102.03002*, 2021.

[13] S. Balaban. Deep learning and face recognition: the state of the art. *Biometric and surveillance technology for human and activity identification XII*, 9457:68–75, 2015.

[14] E. Balas. The prize collecting traveling salesman problem. *Networks*, 19(6):621–636, 1989.

[15] M. L. Balinski and R. E. Quandt. On an integer program for a delivery problem. *Operations research*, 12(2):300–304, 1964.

[16] A. Bdeir, S. Boeder, T. Dernedde, K. Tkachuk, J. K. Falkner, and L. Schmidt-Thieme. Rp-dqn: An application of q-learning to vehicle routing problems. In *German Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 3–16. Springer, 2021.

[17] J. E. Beasley. Route first—cluster second methods for vehicle routing. *Omega*, 11(4):403–408, 1983.

[18] T. Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *omega*, 34(3):209–219, 2006.

[19] R. Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[20] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*. OpenReview.net, 2017.

[21] Y. Bengio, A. Lodi, and A. Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

[22] K. Bestuzheva, M. Besançon, W.-K. Chen, A. Chmiela, T. Donkiewicz, J. van Doornmalen, L. Eifler, O. Gaul, G. Gamrath, A. Gleixner, L. Gottwald, C. Graczyk, K. Halbig, A. Hoen, C. Hojny, R. van der Hulst, T. Koch, M. Lübbecke, S. J. Maher, F. Matter, E. Mühmer, B. Müller, M. E. Pfetsch, D. Rehfeldt, S. Schlein, F. Schlösser, F. Serrano, Y. Shinano, B. Sofranac, M. Turner, S. Vigerske, F. Wegscheider, P. Wellner, D. Weninger, and J. Witzig. The SCIP Optimization Suite 8.0. Technical report, Optimization Online, December 2021.

[23] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, et al. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *The International Journal of High Performance Computing Applications*, 20(4):481–494, 2006.

[24] M. C. Bouzid, H. A. Haddadene, and S. Salhi. An integration of lagrangian split and vns: The case of the capacitated vehicle routing problem. *Computers & Operations Research*, 78:513–525, 2017.

[25] D. Britz, A. Goldie, M.-T. Luong, and Q. Le. Massive exploration of neural machine translation architectures. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1442–1451, Copenhagen, Denmark, Sept. 2017. Association for Computational Linguistics.

[26] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[27] L. I. Burke and J. P. Ignizio. Neural networks and operations research: an overview. *Computers & operations research*, 19(3-4):179–189, 1992.

[28] H. Chen, Y. Jiao, Z. Qin, X. Tang, H. Li, B. An, H. Zhu, and J. Ye. Inbede: Integrating contextual bandit with td learning for joint pricing and dispatch of ride-hailing platforms. In *2019 IEEE International Conference on Data Mining (ICDM)*, pages 61–70. IEEE, 2019.

[29] M. Chen, L. Gao, Q. Chen, and Z. Liu. Dynamic partial removal: A neural network heuristic for large neighborhood search, 2020.

[30] X. Chen and Y. Tian. Learning to perform local rewriting for combinatorial optimization. *Advances in Neural Information Processing Systems*, 32, 2019.

[31] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, 2014.

[32] K. Cho, B. van Merrienboer, C. Gulcehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*, 2014.

[33] J. Christiaens and G. Vanden Berghe. Slack induction by string removals for vehicle routing problems. *Transportation Science*, 54(2):417–433, 2020.

[34] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

[35] N. Christofides and S. Eilon. An algorithm for the vehicle-dispatching problem. *Journal of the Operational Research Society*, 20(3):309–318, 1969.

[36] G. Clarke and J. W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, 12(4):568–581, 1964.

[37] A. Colorni, M. Dorigo, V. Maniezzo, et al. Distributed optimization by ant colonies. In *Proceedings of the first European conference on artificial life*, volume 142, pages 134–142. Paris, France, 1991.

[38] W. Cook. World traveling salesman problem. Available at : `https://www.math.uwaterloo.ca/tsp/world/`, Accessed : 2022-05-10.

[39] J.-F. Cordeau, G. Desaulniers, J. Desrosiers, M. M. Solomon, and F. Soumis. Vrp with time windows. In *The Vehicle Routing Problem*, pages 157–193. SIAM, 2002.

[40] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.

[41] G. B. Dantzig and J. H. Ramser. The truck dispatching problem. *Management science*, 6(1):80–91, 1959.

[42] M. Desrochers, J. Desrosiers, and M. Solomon. A new optimization algorithm for the vehicle routing problem with time windows. *Operations research*, 40(2):342–354, 1992.

[43] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[44] M. Dorigo and T. Stützle. *Ant Colony Optimization for NP-Hard Problems*, pages 153–222. MIT Press, 2004.

[45] I. Drori, B. J. Kates, W. R. Sickinger, A. G. Kharkar, B. Dietrich, A. Shporer, and M. Udell. Galaxytsp: A new billion-node benchmark for tsp. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020.

[46] L. Duan, Y. Zhan, H. Hu, Y. Gong, J. Wei, X. Zhang, and Y. Xu. Efficiently solving the practical vehicle routing problem: A novel joint learning

approach. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 3054–3063, 2020.

[47] Y. Dumas, J. Desrosiers, E. Gelinas, and M. M. Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Operations research*, 43(2):367–371, 1995.

[48] A. Eiben, M. Horvath, W. Kowalczyk, and M. C. Schut. Reinforcement learning for online control of evolutionary algorithms. In *International Workshop on Engineering Self-Organising Applications*, pages 151–160. Springer, 2006.

[49] R. Elshaer and H. Awad. A taxonomic review of metaheuristic algorithms for solving the vehicle routing problem and its variants. *Computers & Industrial Engineering*, 140:106242, 2020.

[50] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

[51] J. K. Falkner and L. Schmidt-Thieme. Learning to solve vehicle routing problems with time windows through joint attention. *arXiv:2006.09100*, 2020.

[52] Y. Fan, M. Ju, C. Zhang, and Y. Ye. Heterogeneous temporal graph neural network. In *Proceedings of the 2022 SIAM International Conference on Data Mining (SDM)*, pages 657–665. SIAM, 2022.

[53] D. Feillet. A tutorial on column generation and branch-and-price for vehicle routing problems. *4or*, 8(4):407–424, 2010.

[54] L. Feng, Y.-S. Ong, A.-H. Tan, and I. W. Tsang. Memes as building blocks: a case study on evolutionary optimization+ transfer learning for routing problems. *Memetic Computing*, 7(3):159–180, 2015.

[55] F. Fernández and M. Veloso. Probabilistic policy reuse in a reinforcement learning agent. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 720–727, 2006.

[56] M. L. Fisher. Optimal solution of vehicle routing problems using minimum k-trees. *Operations research*, 42(4):626–642, 1994.

[57] M. L. Fisher and R. Jaikumar. A generalized assignment heuristic for vehicle routing. *Networks*, 11(2):109–124, 1981.

[58] Z.-H. Fu, K.-B. Qiu, and H. Zha. Generalize a small pre-trained model to arbitrarily large tsp instances. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 7474–7482, 2021.

[59] R. Fukasawa, H. Longo, J. Lysgaard, M. P. d. Aragão, M. Reis, E. Uchoa, and R. F. Werneck. Robust branch-and-cut-and-price for the capacitated vehicle routing problem. *Mathematical programming*, 106(3):491–511, 2006.

[60] R. Gama and H. L. Fernandes. A reinforcement learning approach to the orienteering problem with time windows. *Computers & Operations Research*, 133:105357, 2021.

[61] L. Gao, M. Chen, Q. Chen, G. Luo, N. Zhu, and Z. Liu. Learn to design the heuristics for vehicle routing problem. *arXiv:2002.08539*, 2020.

[62] M. Gendreau, A. Hertz, and G. Laporte. A tabu search heuristic for the vehicle routing problem. *Management science*, 40(10):1276–1290, 1994.

[63] D. George, H. Shen, and E. Huerta. Deep transfer learning: A new deep learning glitch classification method for advanced ligo. *arXiv:1706.07446*, 2017.

[64] B. E. Gillett and L. R. Miller. A heuristic algorithm for the vehicle-dispatch problem. *Operations research*, 22(2):340–349, 1974.

[65] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[66] I. Goodfellow, Y. Bengio, and A. Courville. Regularization for deep learning. *Deep learning*, pages 216–261, 2016.

[67] O. Goudet, C. Grelier, and J.-K. Hao. A deep learning guided memetic framework for graph coloring problems. *Knowledge-Based Systems*, 258:109986, 2022.

[68] A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *arXiv:1410.5401*, 2014.

[69] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[70] W. L. Hamilton. Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(3):1–159, 2022.

[71] D. Hao, D. Zihan, Z. Shanghang, Y. Hang, Z. Hongming, Z. Jingqing, H. Yanhua, Y. Tianyang, Z. Huaqing, and H. Ruitong. *Deep Reinforcement Learning: Fundamentals, Research, and Applications*. Springer Nature, 2020. `http://www.deepreinforcementlearningbook.org`.

[72] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[73] K. Helsgaun. An extension of the lin-kernighan-helsgaun tsp solver for constrained traveling salesman and vehicle routing problems. *Roskilde: Roskilde University*, pages 24–50, 2017.

[74] F. Hernandez, D. Feillet, R. Giroudeau, and O. Naud. Branch-and-price algorithms for the solution of the multi-trip vehicle routing problem with time windows. *European Journal of Operational Research*, 249(2):551–559, 2016.

[75] A. Heuillet, F. Couthouis, and N. Díaz-Rodríguez. Explainability in deep reinforcement learning. *Knowledge-Based Systems*, 214:106685, 2021.

[76] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[77] J. Holler, R. Vuorio, Z. Qin, X. Tang, Y. Jiao, T. Jin, S. Singh, C. Wang, and J. Ye. Deep reinforcement learning for multi-driver vehicle dispatching

and repositioning problem. In *2019 IEEE International Conference on Data Mining (ICDM)*, pages 1090–1095. IEEE, 2019.

[78] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[79] A. Hottung, Y.-D. Kwon, and K. Tierney. Efficient active search for combinatorial optimization problems. In *International Conference on Learning Representations*, 2021.

[80] A. Hottung and K. Tierney. Neural large neighborhood search for the capacitated vehicle routing problem. In *24th European Conference on Artificial Intelligence (ECAI 2020)*, 2020.

[81] J.-T. Huang, J. Li, D. Yu, L. Deng, and Y. Gong. Cross-language knowledge transfer using multilingual deep neural network with shared hidden layers. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 7304–7308. IEEE, 2013.

[82] Y. Huang. Hierarchical reinforcement learning. *Deep Reinforcement Learning: Fundamentals, Research and Applications*, pages 317–333, 2020.

[83] A. Jaiswal, A. R. Babu, M. Z. Zadeh, D. Banerjee, and F. Makedon. A survey on contrastive self-supervised learning. *Technologies*, 9(1):2, 2020.

[84] G. James, D. Witten, T. Hastie, R. Tibshirani, et al. *An introduction to statistical learning*, volume 112. Springer, 2013.

[85] J. Jin, M. Zhou, W. Zhang, M. Li, Z. Guo, Z. Qin, Y. Jiao, X. Tang, C. Wang, J. Wang, et al. Coride: joint order dispatching and fleet management for multi-scale ride-hailing platforms. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 1983–1992, 2019.

[86] D. S. Johnson and L. A. McGeoch. Experimental analysis of heuristics for the stsp. *The traveling salesman problem and its variations*, pages 369–443, 2007.

[87] M. I. Jordan and T. M. Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[88] C. K. Joshi, Q. Cappart, L.-M. Rousseau, and T. Laurent. Learning tsp requires rethinking generalization. In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[89] C. K. Joshi, T. Laurent, and X. Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv:1906.01227*, 2019.

[90] C. K. Joshi, T. Laurent, and X. Bresson. On learning paradigms for the travelling salesman problem. *arXiv:1910.07210*, 2019.

[91] N. Jozefowiez, F. Semet, and E.-G. Talbi. Multi-objective vehicle routing problems. *European journal of operational research*, 189(2):293–309, 2008.

[92] A. A. Juan, C. A. Mendez, J. Faulin, J. De Armas, and S. E. Grasman. Electric vehicles in logistics and transportation: A survey on emerging environmental, strategic, and operational challenges. *Energies*, 9(2):86, 2016.

[93] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30, 2017.

[94] E. B. Khalil, B. Dilkina, G. L. Nemhauser, S. Ahmed, and Y. Shao. Learning to run heuristics in tree search. In *Ijcai*, pages 659–666, 2017.

[95] M. Kim, J. Park, et al. Learning collaborative policies to solve np-hard routing problems. *Advances in Neural Information Processing Systems*, 34:10418–10430, 2021.

[96] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[97] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[98] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[99] W. Kool, H. van Hoof, J. Gromicho, and M. Welling. Deep policy dynamic programming for vehicle routing problems. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 190–213. Springer, 2022.

[100] W. Kool, H. van Hoof, and M. Welling. Attention, learn to solve routing problems! In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[101] N. Kullman and J. Mendoza. Ride-hailing problem. `http://dimacs.rutgers.edu/programs/challenge/vrp/hailing/`, 2022. Accessed on December 20th, 2022.

[102] N. D. Kullman, M. Cousineau, J. C. Goodson, and J. E. Mendoza. Dynamic ride-hailing with electric vehicles. *Transportation Science*, 56(3):775–794, 2022.

[103] Y.-D. Kwon, J. Choo, B. Kim, I. Yoon, Y. Gwon, and S. Min. Pomo: Policy optimization with multiple optima for reinforcement learning. *Advances in Neural Information Processing Systems*, 33:21188–21198, 2020.

[104] J. Kytöjoki, T. Nuortio, O. Bräysy, and M. Gendreau. An efficient variable neighborhood search heuristic for very large scale vehicle routing problems. *Computers & operations research*, 34(9):2743–2757, 2007.

[105] A. Land and A. Doig. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, pages 497–520, 1960.

[106] G. Laporte and Y. Nobert. Exact algorithms for the vehicle routing problem. In *North-Holland Mathematics Studies*, volume 132, pages 147–184. Elsevier, 1987.

[107] G. Laporte, Y. Nobert, and M. Desrochers. Optimal routing under capacity and distance restrictions. *Operations research*, 33(5):1050–1073, 1985.

[108] A. Lazaric. Transfer in reinforcement learning: a framework and a survey. In *Reinforcement Learning*, pages 143–173. Springer, 2012.

[109] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[110] Y. LeCun, C. Cortes, and C. Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[111] K. Lei, P. Guo, Y. Wang, X. Wu, and W. Zhao. Solve routing problems with a residual edge-graph attention neural network. *arXiv:2105.02730*, 2021.

[112] J. K. Lenstra and A. R. Kan. Complexity of vehicle routing and scheduling problems. *Networks*, 11(2):221–227, 1981.

[113] S. Li, Z. Yan, and C. Wu. Learning to delegate for large-scale vehicle routing. *Advances in Neural Information Processing Systems*, 34:26198–26211, 2021.

[114] B. Lin, B. Ghaddar, and J. Nathwani. Deep reinforcement learning for the electric vehicle routing problem with time windows. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[115] S. Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44(10):2245–2269, 1965.

[116] S.-W. Lin, Z.-J. Lee, K.-C. Ying, and C.-Y. Lee. Applying hybrid meta-heuristics for capacitated vehicle routing problem. *Expert Systems with Applications*, 36(2, Part 1):1505–1512, 2009.

[117] H. R. Lourenço, O. C. Martin, and T. Stützle. Iterated local search. In *Handbook of metaheuristics*, pages 320–353. Springer, 2003.

[118] H. Lu, X. Zhang, and S. Yang. A learning-based iterative method for solving vehicle routing problems. In *International conference on learning representations*, 2019.

[119] T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal, Sept. 2015. Association for Computational Linguistics.

[120] J. Lysgaard, A. N. Letchford, and R. W. Eglese. A new branch-and-cut algorithm for the capacitated vehicle routing problem. *Mathematical Programming*, 100(2):423–445, 2004.

[121] Y. Ma, J. Li, Z. Cao, W. Song, L. Zhang, Z. Chen, and J. Tang. Learning to iteratively solve routing problems with dual-aspect collaborative transformer. *Advances in Neural Information Processing Systems*, 34:11096–11107, 2021.

[122] H. H. Mao. A survey on self-supervised pre-training for sequential transfer learning in neural networks. *arXiv:2007.00800*, 2020.

[123] Y. Marinakis. Multiple phase neighborhood search-grasp for the capacitated vehicle routing problem. *Expert Systems with Applications*, 39(8):6807–6815, 2012.

[124] A. Maroti. Rbed: Reward based epsilon decay. *arXiv:1910.13701*, 2019.

[125] V. R. Máximo and M. C. Nascimento. A hybrid adaptive iterated local search with diversification control to the capacitated vehicle routing problem. *European Journal of Operational Research*, 294(3):1108–1119, 2021.

[126] A. Milan, S. H. Rezatofighi, R. Garg, A. Dick, and I. Reid. Data-driven approximations to np-hard problems. In *Thirty-first AAAI conference on artificial intelligence*, 2017.

[127] D. L. Miller. A matching based exact algorithm for capacitated vehicle routing problems. *ORSA Journal on Computing*, 7(1):1–9, 1995.

[128] T. M. Mitchell et al. Machine learning, 1997.

[129] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[130] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[131] V. Nair, S. Bartunov, F. Gimeno, I. von Glehn, P. Lichocki, I. Lobov, B. O'Donoghue, N. Sonnerat, C. Tjandraatmadja, P. Wang, et al. Solving mixed integer programs using neural networks. *arXiv:2012.13349*, 2020.

[132] M. Nazari, A. Oroojlooy, L. Snyder, and M. Takác. Reinforcement learning for solving the vehicle routing problem. *Advances in neural information processing systems*, 31, 2018.

[133] Z. Niu, G. Zhong, and H. Yu. A review on the attention mechanism of deep learning. *Neurocomputing*, 452:48–62, 2021.

[134] K. Oono and T. Suzuki. Graph neural networks exponentially lose expressive power for node classification. In *International Conference on Learning Representations*, 2019.

[135] J. Oren, C. Ross, M. Lefarov, F. Richter, A. Taitler, Z. Feldman, D. Di Castro, and C. Daniel. Solo: search online, learn offline for combinatorial optimization problems. In *Proceedings of the International Symposium on Combinatorial Search*, number 1, pages 97–105, 2021.

[136] I. H. Osman. Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annals of operations research*, 41(4):421–451, 1993.

[137] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. Training language models to

follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[138] H. Paessens. The savings algorithm for the vehicle routing problem. *European Journal of Operational Research*, 34(3):336–344, 1988.

[139] C.-V. Pal and F. Leon. Brief survey of model-based reinforcement learning techniques. In *2020 24th International Conference on System Theory, Control and Computing (ICSTCC)*, pages 92–97. IEEE, 2020.

[140] A. Paliwal, F. Gimeno, V. Nair, Y. Li, M. Lubin, P. Kohli, and O. Vinyals. Regal: Transfer learning for fast optimization of computation graphs. *arXiv:1905.02494*, 2019.

[141] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*, pages 3–4. Courier Corporation, 1998.

[142] A. Parjadis, Q. Cappart, and L.-M. Rousseau. Corailrpc-dqn: a deep reinforcement learning and heuristic approach for the dynamic ride-hailing dimacs implementation challenge. *12th DIMACS Implementation Challenge: Vehicle Routing Problems*, 2022.

[143] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[144] B. Peng, J. Wang, and Z. Zhang. A deep reinforcement learning algorithm using dynamic attention model for vehicle routing problems. In *International Symposium on Intelligence Computation and Applications*, pages 636–650. Springer, 2019.

[145] A. Pessoa, R. Sadykov, E. Uchoa, and F. Vanderbeck. A generic exact solver for vehicle routing and related problems. *Mathematical Programming*, 183(1):483–523, 2020.

[146] V. Pillac, M. Gendreau, C. Guéret, and A. L. Medaglia. A review of dynamic vehicle routing problems. *European Journal of Operational Research*, 225(1):1–11, 2013.

[147] M. Poggi and E. Uchoa. Chapter 3: New exact algorithms for the capacitated vehicle routing problem. In *Vehicle Routing: Problems, Methods, and Applications, Second Edition*, pages 59–86. SIAM, 2014.

[148] J.-Y. Potvin and J.-M. Rousseau. An exchange heuristic for routeing problems with time windows. *Journal of the Operational Research Society*, 46(12):1433–1446, 1995.

[149] C. Prins. A simple and effective evolutionary algorithm for the vehicle routing problem. *Computers & operations research*, 31(12):1985–2002, 2004.

[150] C. Prins, P. Lacomme, and C. Prodhon. Order-first split-second methods for vehicle routing problems: A review. *Transportation Research Part C: Emerging Technologies*, 40:179–200, 2014.

[151] Z. T. Qin, H. Zhu, and J. Ye. Reinforcement learning for ridesharing: An extended survey. *Transportation Research Part C: Emerging Technologies*, 144:103852, 2022.

[152] M. Reimann, K. Doerner, and R. F. Hartl. D-ants: Savings based ants divide and conquer the vehicle routing problem. *Computers & Operations Research*, 31(4):563–591, 2004.

[153] G. Reinelt. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.

[154] L. F. Ribeiro, P. H. Saverese, and D. R. Figueiredo. struc2vec: Learning node representations from structural identity. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 385–394, 2017.

[155] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[156] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein. Temporal graph networks for deep learning on dynamic graphs. *arXiv:2006.10637*, 2020.

[157] D. M. Ryan, C. Hjorring, and F. Glover. Extensions of the petal method for vehicle routeing. *Journal of the Operational Research Society*, 44(3):289–296, 1993.

[158] M. Saint-Guillain, Y. Deville, and C. Solnon. The dial-a-ride problem with transfers and stochastic customers. In *Doctoral program of the 20th International Conference of Principles and Practice of Constraint Programming (CP)*, pages 72–78, 2014.

[159] T. Schneider. Taxi and ridehailing app usage in new york city. `https://toddwschneider.com/dashboards/nyc-taxi-ridehailing-uber-lyft-data/`, 2022. [Accessed 20-Dec-2022].

[160] A. Schrijver. On the history of combinatorial optimization (till 1960). *Handbooks in operations research and management science*, 12:1–68, 2005.

[161] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[162] A. Sharif Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 806–813, 2014.

[163] Y. Shi, Z. Huang, S. Feng, H. Zhong, W. Wang, and Y. Sun. Masked label prediction: Unified message passing model for semi-supervised classification. In Z. Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 1548–1554. ijcai.org, 2021.

[164] Z. Shou and X. Di. Multi-agent reinforcement learning for dynamic routing games: A unified paradigm. *arXiv:2011.10915*, 2020.

[165] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[166] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[167] R. A. Skitt and R. R. Levary. Vehicle routing via column generation. *European Journal of Operational Research*, 21(1):65–76, 1985.

[168] K. A. Smith. Neural networks for combinatorial optimization: a review of more than a decade of research. *INFORMS Journal on Computing*, 11(1):15–34, 1999.

[169] K. Sörensen and P. Schittekat. Statistical analysis of distance-based path relinking for the capacitated vehicle routing problem. *Computers & Operations Research*, 40(12):3197–3205, 2013.

[170] A. Subramanian, E. Uchoa, and L. S. Ochi. A hybrid algorithm for a class of vehicle routing problems. *Computers & Operations Research*, 40(10):2519–2531, 2013.

[171] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[172] A. A. Syed, K. Akhnoukh, B. Kaltenhaeuser, and K. Bogenberger. Neural network based large neighborhood search algorithm for ride hailing services. In *EPIA Conference on Artificial Intelligence*, pages 584–595. Springer, 2019.

[173] E.-G. Talbi. *Metaheuristics: from design to implementation*. John Wiley & Sons, 2009.

[174] M. Tyagi. A practical method for the truck dispatching problem. *Journal of the Operations Research Society of Japan*, 10:76–92, 1968.

[175] E. Uchoa, D. Pecin, A. Pessoa, M. Poggi, T. Vidal, and A. Subramanian. New benchmark instances for the capacitated vehicle routing problem. *European Journal of Operational Research*, 257(3):845–858, 2017.

[176] A. Van Breedam. A parametric analysis of heuristics for the vehicle routing problem with side-constraints. *European Journal of Operational Research*, 137(2):348–370, 2002.

[177] M. Van Knippenberg, M. Holenderski, and V. Menkovski. Complex vehicle routing with memory augmented neural networks. In *2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)*, volume 1, pages 303–308. IEEE, 2020.

[178] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[179] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[180] J. M. Vera and A. G. Abad. Deep reinforcement learning for routing a heterogeneous fleet of vehicles. In *2019 IEEE Latin American Conference on Computational Intelligence (LA-CCI)*, pages 1–6. IEEE, 2019.

[181] T. Vidal. Hybrid genetic search for the cvrp: Open-source implementation and swap* neighborhood. *Computers & Operations Research*, 140:105643, 2022.

[182] T. Vidal, T. G. Crainic, M. Gendreau, N. Lahrichi, and W. Rei. A hybrid genetic algorithm for multidepot and periodic vehicle routing problems. *Operations Research*, 60(3):611–624, 2012.

[183] T. Vidal, T. G. Crainic, M. Gendreau, and C. Prins. A hybrid genetic algorithm with adaptive diversity management for a large class of vehicle routing problems with time-windows. *Computers & operations research*, 40(1):475–489, 2013.

[184] T. Vidal, G. Laporte, and P. Matl. A concise guide to existing and emerging vehicle routing problem variants. *European Journal of Operational Research*, 286(2):401–416, 2020.

[185] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. *Advances in neural information processing systems*, 28, 2015.

[186] Q. Wang, Y. Hao, and J. Cao. Learning to traverse over graphs with a monte carlo tree search-based self-play framework. *Engineering Applications of Artificial Intelligence*, 105:104422, 2021.

[187] Z. Wang, Z. Dai, B. Póczos, and J. Carbonell. Characterizing and avoiding negative transfer. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11293–11302, 2019.

[188] Z. Wang, Z. Qin, X. Tang, J. Ye, and H. Zhu. Deep reinforcement learning with knowledge transfer for online rides order dispatching. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 617–626. IEEE, 2018.

[189] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[190] S. Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9(1):11–12, 1962.

[191] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[192] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

[193] R. S. Woodworth and E. Thorndike. The influence of improvement in one mental function upon the efficiency of other functions.(i). *Psychological review*, 8(3):247, 1901.

[194] L. Wu, P. Cui, J. Pei, and L. Zhao. Graph neural networks: Foundations, frontiers, and applications, 2022.

[195] Y. Wu, W. Song, Z. Cao, J. Zhang, and A. Lim. Learning improvement heuristics for solving routing problems.. *IEEE transactions on neural networks and learning systems*, 2021.

[196] L. Xin, W. Song, Z. Cao, and J. Zhang. Step-wise deep learning models for solving routing problems. *IEEE Transactions on Industrial Informatics*, 17(7):4861–4871, 2020.

[197] L. Xin, W. Song, Z. Cao, and J. Zhang. Multi-decoder attention model with embedding glimpse for solving vehicle routing problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, number 13, pages 12042–12049, 2021.

[198] Y. Xu, M. Fang, L. Chen, G. Xu, Y. Du, and C. Zhang. Reinforcement learning with multiple relational attention for solving vehicle routing problems. *IEEE Transactions on Cybernetics*, 2021.

[199] Z. Xu, Z. Li, Q. Guan, D. Zhang, Q. Li, J. Nan, C. Liu, W. Bian, and J. Ye. Large-scale order dispatch in on-demand ride-hailing platforms: A learning and planning approach. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 905–913, 2018.

[200] B. Yu, Z.-Z. Yang, and B. Yao. An improved ant colony optimization for vehicle routing problem. *European Journal of Operational Research*, 196(1):171–176, 2009.

[201] W. Yu, J. Tan, C. K. Liu, and G. Turk. Preparing for the unknown: Learning a universal policy with online system identification. *arXiv:1702.02453*, 2017.

[202] E. Yuan and P. Van Hentenryck. Learning model predictive controllers for real-time ride-hailing vehicle relocation and pricing decisions. *arXiv:2111.03204*, 2021.

[203] S. Zajac and S. Huber. Objectives and methods in multi-objective routing problems: a survey and classification scheme. *European Journal of Operational Research*, 290(1):1–25, 2021.

[204] F. Zantalis, G. Koulouras, S. Karabetsos, and D. Kandris. A review of machine learning and iot in smart transportation. *Future Internet*, 11(4):94, 2019.

[205] Y. Zhang and Q. Yang. An overview of multi-task learning. *National Science Review*, 5(1):30–43, 2018.

[206] J. Zhao, M. Mao, X. Zhao, and J. Zou. A hybrid of deep reinforcement learning and local search for the vehicle routing problems. *IEEE Transactions on Intelligent Transportation Systems*, 22(11):7208–7218, 2020.

[207] J. Zhao, M. Mao, X. Zhao, and J. Zou. A hybrid of deep reinforcement learning and local search for the vehicle routing problems. *IEEE Transactions on Intelligent Transportation Systems*, 22:7208–7218, 2021.

[208] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[209] M. Zhou, J. Jin, W. Zhang, Z. Qin, Y. Jiao, C. Wang, G. Wu, Y. Yu, and J. Ye. Multi-agent reinforcement learning for order-dispatching via order-vehicle distribution matching. In *Proceedings of the 28th ACM international conference on information and knowledge management*, pages 2645–2653, 2019.

[210] Z. Zhu, K. Lin, and J. Zhou. Transfer learning in deep reinforcement learning: A survey. *arXiv:2009.07888*, 2020.

Publications

International conferences

Yaddaden, A., Harispe, S., & Vasquez, M. (2022). Is Transfer Learning Helpful for Neural Combinatorial Optimization Applied to Vehicle Routing Problems?. Computing and Informatics, 41(1), 172-190.

Yaddaden, A., Harispe, S., & Vasquez, M. (2022, July). Neural Order-First Split-Second Algorithm for the Capacitated Vehicle Routing Problem. In International Conference on Optimization and Learning (pp. 168-185). Cham: Springer International Publishing.

National conferences

Yaddaden, A., Harispe, S., Vasquez, M., & Buljubašic, M. (2020, June). Apprentissage Automatique pour l'Optimisation Combinatoire: Étude du problème du voyageur de commerce. CNIA 2020 - Conférence Nationale en Intelligence Artificielle.

Yaddaden, A., Harispe, S., & Vasquez, M. (2021, April). Apprentissage par transfert: du TSP au VRP. In ROADEF 2021-22e congrès annuel de la société Française de Recherche Opérationnelle et d'Aide à la Décision.

Yaddaden, A., Harispe, S., & Vasquez, M. (2022, February). Évaluation empirique des modèles d'apprentissage profond pour le problème de tournées de véhicules

avec contrainte de capacité. In ROADEF 2022-23ème congrès annuel de la Société Française de Recherche Opérationnelle et d'Aide à la Décision.

Yaddaden, A., Harispe, S., & Vasquez, M. (2023, February). Une méthode à base d'apprentissage par renforcement pour le problème de tournées de véhicules avec contrainte de capacité. In ROADEF 2023-24ème édition du congrès annuel de la Société Française de Recherche Opérationnelle et d'Aide à la Décision.

Appendices

A First evaluation results on CVRPLib

Q	Vehicle's capacity
K	Number of routes
Tightness	$\frac{\sum_{i=1}^{n} q_i}{KQ}$
$q_{\min}$	The minimum demand
$q_{\max}$	The maximum demand
BKS	HGS solution
$obj.$	Solution objective value
gap	%gap to the BKS $((1 - \frac{obj.}{BKS}) \cdot 100)$
t_{best}	cpu time to find $obj.$ in minutes
t_{best}	total runtime in minutes

Table 7: Description of table columns.

Instance	Q	K	Tightness	q_{min}	q_{max}	BKS	obj.	%gap	t_{tot}	t_{best}
X-n101-k25	206	25	1.0	1	100	27591.0	51464	86.52	469.46	445.97
X-n106-k14	600	14	0.94	50	100	26364.0	45597	72.95	437.06	427.97
X-n110-k13	66	13	0.95	5	10	14971.0	15762	5.28	448.09	442.76
X-n115-k10	169	10	0.91	1	99	12747.0	49745	290.25	470.04	148.05
X-n120-k6	21	6	0.94	1	1	13332.0	14891	11.69	458.89	393.51
X-n125-k30	188	30	0.98	1	100	55539.0	61208	10.21	569.52	558.33
X-n129-k18	39	18	0.95	1	10	28940.0	30101	4.01	533.82	527.10
X-n134-k13	643	13	0.98	3	100	10916.0	24500	124.44	542.52	160.41
X-n139-k10	106	10	0.98	5	10	13590.0	14478	6.53	546.91	540.45
X-n143-k7	1190	7	0.9	2	99	15700.0	70276	347.62	552.03	83.43
X-n148-k46	18	46	0.99	1	10	43448.0	43844	0.91	710.73	670.28
X-n153-k22	144	22	0.97	1	98	21225.0	30155	42.07	655.73	327.78
X-n157-k13	12	13	1.0	1	1	16876.0	17577	4.15	624.01	607.78
X-n162-k11	1174	11	0.94	51	100	14138.0	20887	47.74	652.78	649.93
X-n167-k10	133	10	0.93	5	10	20557.0	22018	7.11	655.23	625.14
X-n172-k51	161	51	0.99	2	100	45607.0	84082	84.36	830.16	561.60
X-n176-k26	142	26	0.98	1	100	47812.0	84384	76.49	747.93	534.73
X-n181-k23	8	23	0.98	1	1	25569.0	26628	4.14	751.62	746.37
X-n186-k15	974	15	0.95	50	100	24145.0	93405	286.85	760.40	194.07
X-n190-k8	138	8	0.94	1	10	16980.0	18401	8.37	733.11	649.66
X-n195-k51	181	51	1.0	1	100	44225.0	97315	120.05	877.35	443.81
X-n200-k36	402	36	0.99	2	100	58578.0	73801	25.99	868.21	868.12
X-n204-k19	836	19	0.95	50	100	19565.0	83074	324.61	855.81	151.39
X-n209-k16	101	16	0.96	5	10	30656.0	33770	10.16	836.65	806.56
X-n214-k11	944	11	1.0	2	100	10856.0	47658	339.00	838.79	563.72
X-n219-k73	3	73	1.0	1	1	117595.0	118760	0.99	1075.14	1038.61
X-n223-k34	37	34	0.98	1	10	40437.0	43165	6.75	961.39	843.33
X-n228-k23	154	23	0.98	1	100	25742.0	31425	22.08	942.25	887.06
X-n233-k16	631	16	1.0	1	100	19230.0	97499	407.02	955.05	198.59
X-n237-k14	18	14	0.94	1	1	27042.0	29933	10.69	943.16	881.18
X-n242-k48	28	48	0.99	1	10	82771.0	86007	3.91	1086.66	1065.88

Table 8: NCO-AM solution cost vs. HGS solution cost on the set X.

Instance	Q	K	Tightness	q_{min}	q_{max}	BKS	obj.	%gap	t_{tot}	t_{best}
A-n32-k5	100	5	0.82	1	24	784.0	787	0.38	83.56	19.83
A-n33-k5	100	5	0.89	2	24	661.0	737	11.50	85.76	72.10
A-n33-k6	100	6	0.9	1	66	742.0	792	6.74	89.03	50.92
A-n34-k5	100	5	0.92	1	25	778.0	796	2.31	88.33	14.71
A-n36-k5	100	5	0.88	1	23	799.0	826	3.38	91.92	35.76
A-n37-k5	100	5	0.81	1	27	669.0	759	13.45	93.94	46.98
A-n37-k6	100	6	0.95	1	66	949.0	961	1.26	98.31	79.96
A-n38-k5	100	5	0.96	1	26	730.0	759	3.97	97.51	20.70
A-n39-k5	100	5	0.95	1	26	822.0	864	5.11	101.80	20.80
A-n39-k6	100	6	0.88	1	72	831.0	986	18.65	102.81	59.80
A-n44-k6	100	6	0.95	2	24	937.0	960	2.45	112.88	42.99
A-n45-k6	100	6	0.99	2	24	944.0	993	5.19	115.45	104.66
A-n45-k7	100	7	0.91	1	26	1146.0	1171	2.18	115.07	40.42
A-n46-k7	100	7	0.86	1	26	914.0	943	3.17	117.44	38.67
A-n48-k7	100	7	0.89	2	26	1073.0	1093	1.86	120.92	80.67
A-n53-k7	100	7	0.95	1	30	1010.0	1064	5.35	132.33	47.06
A-n54-k7	100	7	0.96	2	36	1167.0	1248	6.94	133.83	42.70
A-n55-k9	100	9	0.93	2	66	1073.0	1197	11.56	143.04	137.95
A-n60-k9	100	9	0.92	1	48	1354.0	1406	3.84	152.98	117.98
A-n61-k9	100	9	0.98	2	72	1034.0	1328	28.43	156.00	155.19
A-n62-k8	100	8	0.92	1	26	1288.0	1345	4.43	152.78	124.13
A-n63-k10	100	10	0.93	1	63	1314.0	1408	7.15	164.15	163.94
A-n63-k9	100	9	0.97	1	26	1616.0	1668	3.22	159.94	113.83
A-n64-k9	100	9	0.94	1	54	1401.0	1507	7.57	163.95	94.28
A-n65-k9	100	9	0.97	1	26	1174.0	1191	1.45	165.79	124.21
A-n69-k9	100	9	0.94	1	39	1159.0	1189	2.59	177.08	140.76
A-n80-k10	100	10	0.94	1	26	1763.0	1812	2.78	204.86	196.44

Table 9: NCO-AM solution cost vs. HGS solution cost on the set A.

Instance	Q	K	Tightness	q_{min}	q_{max}	BKS	obj.	%gap	t_{tot}	t_{best}
B-n31-k5	100	5	0.82	2	25	672.0	695	3.42	81.33	18.41
B-n34-k5	100	5	0.91	1	69	788.0	848	7.61	89.30	52.00
B-n35-k5	100	5	0.87	1	26	955.0	989	3.56	91.36	80.68
B-n38-k6	100	6	0.85	1	26	805.0	820	1.86	98.16	86.71
B-n39-k5	100	5	0.88	1	25	549.0	591	7.65	98.62	51.89
B-n41-k6	100	6	0.94	6	25	829.0	875	5.55	104.79	66.11
B-n43-k6	100	6	0.87	1	25	742.0	749	0.94	108.83	62.95
B-n44-k7	100	7	0.92	3	69	909.0	939	3.30	115.35	86.22
B-n45-k5	100	5	0.97	1	25	751.0	780	3.86	114.34	35.90
B-n45-k6	100	6	0.99	2	26	678.0	694	2.36	114.97	78.99
B-n50-k7	100	7	0.87	2	63	741.0	769	3.78	127.16	123.67
B-n50-k8	100	8	0.92	2	69	1312.0	1364	3.96	129.21	120.56
B-n51-k7	100	7	0.98	3	42	1016.0	1046	2.95	129.40	121.12
B-n52-k7	100	7	0.87	1	26	747.0	779	4.28	135.11	66.52
B-n56-k7	100	7	0.88	1	26	707.0	736	4.10	138.41	136.20
B-n57-k7	100	7	1.0	1	60	1153.0	1209	4.86	143.27	142.57
B-n57-k9	100	9	0.89	2	26	1598.0	1652	3.38	144.04	115.19
B-n63-k10	100	10	0.92	2	48	1496.0	1538	2.81	161.95	160.76
B-n64-k9	100	9	0.98	1	54	861.0	934	8.48	163.71	162.70
B-n66-k9	100	9	0.96	1	23	1316.0	1369	4.03	167.12	164.83
B-n67-k10	100	10	0.91	1	26	1032.0	1109	7.46	172.03	157.97
B-n68-k9	100	9	0.93	1	48	1272.0	1336	5.03	173.97	96.96
B-n78-k10	100	10	0.94	1	26	1221.0	1302	6.63	201.93	199.88

Table 10: NCO-AM solution cost vs. HGS solution cost on the set B.

Instance	Q	K	Tightness	q_{min}	q_{max}	BKS	obj.	%gap	t_{tot}	t_{best}
E-n101-k14	112	14	0.93	1	41	1067.0	1182	10.78	423.10	417.99
E-n101-k8	200	8	0.91	1	41	815.0	847	3.93	400.91	345.38
E-n22-k4	6000	4	0.94	100	2500	375.0	375	0.00	63.04	2.96
E-n23-k3	4500	3	0.75	60	4100	569.0	577	1.40	59.22	2.82
E-n30-k3	4500	3	0.94	100	3100	503.0	513	1.99	79.96	64.76
E-n33-k4	8000	4	0.92	40	4000	835.0	848	1.56	84.40	41.99
E-n51-k5	160	5	0.97	3	41	521.0	538	3.26	126.28	72.63
E-n76-k10	140	10	0.97	1	37	830.0	847	2.05	202.38	192.71
E-n76-k14	100	14	0.97	1	37	1021.0	1044	2.25	211.48	209.19
E-n76-k7	220	7	0.89	1	37	682.0	713	4.55	192.90	165.82
E-n76-k8	180	8	0.95	1	37	735.0	752	2.31	193.44	148.52

Table 11: NCO-AM solution cost vs. HGS solution cost on the set E.

Instance	Q	K	Tightness	q_{min}	q_{max}	BKS	obj.	%gap	t_{tot}	t_{best}
F-n135-k7	2210	7	0.95	1	1126	1162.0	1329	14.37	531.73	77.84
F-n45-k4	2010	4	0.9	1	1300	724.0	798	10.22	113.27	56.53
F-n72-k4	30000	4	0.96	4	21611	237.0	251	5.91	174.28	171.95

Table 12: NCO-AM solution cost vs. HGS solution cost on the set F

Instance	Q	K	Tightness	q_{min}	q_{max}	BKS	obj.	%gap	t_{tot}	t_{best}
M-n101-k10	200	10	0.91	10	50	820.0	852	3.90	413.03	105.53
M-n121-k7	200	7	0.98	2	35	1034.0	1158	11.99	482.23	179.65
M-n151-k12	200	12	0.93	1	41	1015.0	1088	7.19	604.60	603.34
M-n200-k17	200	17	0.94	1	41	1282.0	1365	6.47	880.79	621.35

Table 13: NCO-AM solution cost vs. HGS solution cost on the set M.

Instance	Q	K	Tightness	q_{min}	q_{max}	BKS	obj.	%gap	t_{tot}	t_{best}
P-n101-k4	400	4	0.91	1	41	681.0	727	6.75	399.17	40.09
P-n16-k8	35	8	0.88	6	31	450.0	450	0.00	54.59	1.69
P-n19-k2	160	2	0.97	6	31	212.0	215	1.42	49.08	19.38
P-n20-k2	160	2	0.97	6	31	216.0	219	1.39	50.96	4.30
P-n21-k2	160	2	0.93	6	30	211.0	217	2.84	53.51	7.87
P-n22-k2	160	2	0.96	6	30	216.0	216	0.00	55.54	28.67
P-n22-k8	3000	8	0.94	100	2500	590.0	604	2.37	58.47	1.29
P-n23-k8	40	8	0.98	5	30	529.0	532	0.57	71.16	6.67
P-n40-k5	140	5	0.88	3	41	458.0	486	6.11	101.83	51.29
P-n45-k5	150	5	0.92	3	41	510.0	524	2.75	112.14	39.56
P-n50-k10	100	10	0.95	5	37	696.0	730	4.89	133.69	76.52
P-n50-k7	150	7	0.91	5	37	554.0	571	3.07	127.92	64.93
P-n50-k8	120	8	0.99	5	37	631.0	637	0.95	130.42	42.61
P-n51-k10	80	10	0.97	3	41	741.0	751	1.35	137.82	64.22
P-n55-k10	115	10	0.91	5	37	694.0	709	2.16	145.23	119.72
P-n55-k15	70	15	0.99	5	37	941.0	952	1.17	158.69	122.15
P-n55-k7	170	7	0.88	5	37	568.0	585	2.99	138.53	84.34
P-n55-k8	160	8	0.81	5	37	576.0	600	4.17	136.43	77.26
P-n60-k10	120	10	0.94	5	37	744.0	757	1.75	156.04	106.44
P-n60-k15	80	15	0.94	5	37	968.0	1013	4.65	167.94	135.39
P-n65-k10	130	10	0.94	5	37	792.0	812	2.53	168.00	130.79
P-n70-k10	135	10	0.97	5	37	827.0	838	1.33	185.68	134.82
P-n76-k4	350	4	0.97	1	37	593.0	641	8.09	182.45	178.01
P-n76-k5	280	5	0.97	1	37	627.0	647	3.19	186.08	181.17

Table 14: NCO-AM solution cost vs. HGS solution cost on the set P.

B Results of NOFSS on CVRPLib

Instance	Q	K	Tightness	q_{min}	q_{max}	BKS	obj.	gap	t_{tot}	t_{best}
X-n101-k25	206	25	1.0	1	100	27591.0	28738	4.16%	64.49	37.70
X-n106-k14	600	14	0.94	50	100	26364.0	27152	2.99%	68.41	56.87
X-n110-k13	66	13	0.95	5	10	14971.0	15461	3.27%	70.63	32.85
X-n115-k10	169	10	0.91	1	99	12747.0	13255	3.99%	74.00	35.92
X-n120-k6	21	6	0.94	1	1	13332.0	13955	4.67%	77.75	71.48
X-n125-k30	188	30	0.98	1	100	55539.0	58735	5.75%	82.59	78.38
X-n129-k18	39	18	0.95	1	10	28940.0	30689	6.04%	59.00	48.88
X-n134-k13	643	13	0.98	3	100	10916.0	11658	6.80%	62.43	57.39
X-n139-k10	106	10	0.98	5	10	13590.0	14217	4.61%	65.74	43.96
X-n143-k7	1190	7	0.9	2	99	15700.0	16935	7.87%	68.41	68.08
X-n148-k46	18	46	0.99	1	10	43448.0	45288	4.23%	71.69	68.90
X-n153-k22	144	22	0.97	1	98	21225.0	23265	9.61%	74.41	72.08
X-n157-k13	12	13	1.0	1	1	16876.0	17668	4.69%	78.08	77.56
X-n162-k11	1174	11	0.94	51	100	14138.0	15023	6.26%	82.68	77.77
X-n167-k10	133	10	0.93	5	10	20557.0	22551	9.70%	87.24	71.10
X-n172-k51	161	51	0.99	2	100	45607.0	47457	4.06%	91.21	83.02
X-n176-k26	142	26	0.98	1	100	47812.0	51859	8.46%	93.15	91.89
X-n181-k23	8	23	0.98	1	1	25569.0	26344	3.03%	96.52	88.48
X-n186-k15	974	15	0.95	50	100	24145.0	26074	7.99%	101.44	82.99
X-n190-k8	138	8	0.94	1	10	16980.0	18337	7.99%	105.51	86.66
X-n195-k51	181	51	1.0	1	100	44225.0	46759	5.73%	109.37	98.73
X-n200-k36	402	36	0.99	2	100	58578.0	63347	8.14%	112.79	100.43
X-n204-k19	836	19	0.95	50	100	19565.0	21260	8.66%	157.01	133.28
X-n209-k16	101	16	0.96	5	10	30656.0	33204	8.31%	162.38	153.91
X-n214-k11	944	11	1.0	2	100	10856.0	12144	11.86%	167.90	162.58
X-n219-k73	3	73	1.0	1	1	117595.0	118710	0.95%	172.96	168.39
X-n223-k34	37	34	0.98	1	10	40437.0	42842	5.95%	177.48	122.05
X-n228-k23	154	23	0.98	1	100	25742.0	28003	8.78%	183.04	172.29

X-n233-k16	631	16	1.0	1	100	19230.0	21114	9.80%	190.33	185.17
X-n237-k14	18	14	0.94	1	1	27042.0	29504	9.10%	198.09	169.85
X-n242-k48	28	48	0.99	1	10	82771.0	89334	7.93%	201.62	200.40

Table 15: NOFSS solution cost vs. HGS solution cost on the set X.

Instance	Q	K	Tightness	q_{min}	q_{max}	BKS	obj.	gap	t_{tot}	t_{best}
A-n32-k5	100	5	0.82	1	24	784.0	788	0.51%	64.89	24.76
A-n33-k5	100	5	0.89	2	24	661.0	661	0.00%	37.05	1.32
A-n33-k6	100	6	0.9	1	66	742.0	744	0.27%	36.77	6.99
A-n34-k5	100	5	0.92	1	25	778.0	800	2.83%	38.06	0.65
A-n36-k5	100	5	0.88	1	23	799.0	818	2.38%	39.86	9.70
A-n37-k5	100	5	0.81	1	27	669.0	693	3.59%	40.91	0.78
A-n37-k6	100	6	0.95	1	66	949.0	969	2.11%	41.05	0.19
A-n38-k5	100	5	0.96	1	26	730.0	750	2.74%	42.25	7.90
A-n39-k5	100	5	0.95	1	26	822.0	856	4.14%	42.81	1.39
A-n39-k6	100	6	0.88	1	72	831.0	845	1.68%	42.51	7.16
A-n44-k6	100	6	0.95	2	24	937.0	970	3.52%	47.63	35.33
A-n45-k6	100	6	0.99	2	24	944.0	971	2.86%	48.38	2.33
A-n45-k7	100	7	0.91	1	26	1146.0	1168	1.92%	48.32	30.72
A-n46-k7	100	7	0.86	1	26	914.0	969	6.02%	49.22	7.76
A-n48-k7	100	7	0.89	2	26	1073.0	1103	2.80%	50.89	3.63
A-n53-k7	100	7	0.95	1	30	1010.0	1037	2.67%	55.80	6.28
A-n54-k7	100	7	0.96	2	36	1167.0	1219	4.46%	57.42	12.33
A-n55-k9	100	9	0.93	2	66	1073.0	1079	0.56%	58.13	34.71
A-n60-k9	100	9	0.92	1	48	1354.0	1416	4.58%	63.38	1.81
A-n61-k9	100	9	0.98	2	72	1034.0	1070	3.48%	64.19	36.39
A-n62-k8	100	8	0.92	1	26	1288.0	1346	4.50%	64.84	14.60
A-n63-k10	100	10	0.93	1	63	1314.0	1357	3.27%	65.69	24.26
A-n63-k9	100	9	0.97	1	26	1616.0	1685	4.27%	65.92	25.50
A-n64-k9	100	9	0.94	1	54	1401.0	1443	3.00%	66.71	13.09
A-n65-k9	100	9	0.97	1	26	1174.0	1189	1.28%	41.03	3.21
A-n69-k9	100	9	0.94	1	39	1159.0	1188	2.50%	44.56	6.94
A-n80-k10	100	10	0.94	1	26	1763.0	1865	5.79%	50.99	29.91

Table 16: NOFSS solution cost vs. HGS solution cost on the set A.

Instance	Q	K	Tightness	q_{min}	q_{max}	BKS	obj.	gap	t_{tot}	t_{best}
B-n31-k5	100	5	0.82	2	25	672.0	672	0.00%	68.37	0.58
B-n34-k5	100	5	0.91	1	69	788.0	804	2.03%	40.00	3.35
B-n35-k5	100	5	0.87	1	26	955.0	962	0.73%	39.78	3.21
B-n38-k6	100	6	0.85	1	26	805.0	815	1.24%	42.47	5.06
B-n39-k5	100	5	0.88	1	25	549.0	558	1.64%	43.40	32.54
B-n41-k6	100	6	0.94	6	25	829.0	838	1.09%	45.30	14.81
B-n43-k6	100	6	0.87	1	25	742.0	756	1.89%	47.25	12.28
B-n44-k7	100	7	0.92	3	69	909.0	919	1.10%	48.19	17.94
B-n45-k5	100	5	0.97	1	25	751.0	766	2.00%	49.26	38.40
B-n45-k6	100	6	0.99	2	26	678.0	722	6.49%	48.99	7.84
B-n50-k7	100	7	0.87	2	63	741.0	743	0.27%	54.03	5.95
B-n50-k8	100	8	0.92	2	69	1312.0	1340	2.13%	53.73	26.85
B-n51-k7	100	7	0.98	3	42	1016.0	1019	0.29%	54.79	33.06
B-n52-k7	100	7	0.87	1	26	747.0	757	1.34%	55.76	9.25
B-n56-k7	100	7	0.88	1	26	707.0	730	3.25%	59.98	14.86
B-n57-k7	100	7	1.0	1	60	1153.0	1154	0.09%	60.80	59.54
B-n57-k9	100	9	0.89	2	26	1598.0	1640	2.63%	61.42	25.91
B-n63-k10	100	10	0.92	2	48	1496.0	1592	6.42%	66.30	38.72
B-n64-k9	100	9	0.98	1	54	861.0	910	5.69%	66.84	62.89
B-n66-k9	100	9	0.96	1	23	1316.0	1354	2.89%	41.43	21.50
B-n67-k10	100	10	0.91	1	26	1032.0	1097	6.30%	42.02	9.85
B-n68-k9	100	9	0.93	1	48	1272.0	1298	2.04%	42.61	37.38
B-n78-k10	100	10	0.94	1	26	1221.0	1282	5.00%	48.98	44.44

Table 17: NOFSS solution cost vs. HGS solution cost on the set B

Instance	Q	K	Tightness	q_{min}	q_{max}	BKS	obj.	gap	t_{tot}	t_{best}
E-n101-k14	112	14	0.93	1	41	1067.0	1108	3.84%	64.57	32.25
E-n101-k8	200	8	0.91	1	41	815.0	859	5.40%	64.49	12.28
E-n22-k4	6000	4	0.94	100	2500	375.0	375	0.00%	47.43	3.92
E-n23-k3	4500	3	0.75	60	4100	569.0	569	0.00%	49.00	46.38
E-n30-k3	4500	3	0.94	100	3100	503.0	503	0.00%	60.13	0.68
E-n33-k4	8000	4	0.92	40	4000	835.0	866	3.71%	36.56	33.21
E-n51-k5	160	5	0.97	3	41	521.0	544	4.41%	53.20	1.94
E-n76-k10	140	10	0.97	1	37	830.0	857	3.25%	47.24	10.20
E-n76-k14	100	14	0.97	1	37	1021.0	1066	4.41%	47.34	6.45
E-n76-k7	220	7	0.89	1	37	682.0	706	3.52%	47.50	5.94
E-n76-k8	180	8	0.95	1	37	735.0	761	3.54%	47.63	15.43

Table 18: NOFSS solution cost vs. HGS solution cost on the set E.

Instance	Q	K	Tightness	q_{min}	q_{max}	BKS	obj.	gap	t_{tot}	t_{best}
F-n135-k7	2210	7	0.95	1	1126	1162.0	1235	6.28%	64.86	44.91
F-n45-k4	2010	4	0.9	1	1300	724.0	730	0.83%	49.49	15.66
F-n72-k4	30000	4	0.96	4	21611	237.0	260	9.70%	45.76	1.75

Table 19: NOFSS solution cost vs. HGS solution cost on the set F.

Instance	Q	K	Tightness	q_{min}	q_{max}	BKS	obj.	gap	t_{tot}	t_{best}
M-n101-k10	200	10	0.91	10	50	820.0	834	1.71%	62.86	37.59
M-n121-k7	200	7	0.98	2	35	1034.0	1068	3.29%	77.37	77.26
M-n151-k12	200	12	0.93	1	41	1015.0	1066	5.02%	72.00	70.01
M-n200-k17	200	17	0.94	1	41	1282.0	1344	4.84%	107.81	101.36

Table 20: NOFSS solution cost vs. HGS solution cost on the set M.

Instance	Q	K	Tightness	q_{min}	q_{max}	BKS	obj.	gap	t_{tot}	t_{best}
P-n101-k4	400	4	0.91	1	41	681.0	688	1.03%	66.12	28.25
P-n16-k8	35	8	0.88	6	31	450.0	450	0.00%	41.11	0.02
P-n19-k2	160	2	0.97	6	31	212.0	219	3.30%	46.35	6.28
P-n20-k2	160	2	0.97	6	31	216.0	219	1.39%	48.15	8.91
P-n21-k2	160	2	0.93	6	30	211.0	218	3.32%	49.61	26.29
P-n22-k2	160	2	0.96	6	30	216.0	224	3.70%	50.04	3.28
P-n22-k8	3000	8	0.94	100	2500	590.0	590	0.00%	49.91	10.14
P-n23-k8	40	8	0.98	5	30	529.0	536	1.32%	51.51	0.05
P-n40-k5	140	5	0.88	3	41	458.0	465	1.53%	44.66	0.13
P-n45-k5	150	5	0.92	3	41	510.0	514	0.78%	49.47	0.71
P-n50-k10	100	10	0.95	5	37	696.0	718	3.16%	54.42	13.29
P-n50-k7	150	7	0.91	5	37	554.0	575	3.79%	54.48	0.45
P-n50-k8	120	8	0.99	5	37	631.0	644	2.06%	54.52	22.09
P-n51-k10	80	10	0.97	3	41	741.0	758	2.29%	55.14	1.36
P-n55-k10	115	10	0.91	5	37	694.0	724	4.32%	58.87	20.47
P-n55-k15	70	15	0.99	5	37	941.0	968	2.87%	58.53	3.53
P-n55-k7	170	7	0.88	5	37	568.0	597	5.11%	59.36	0.70
P-n55-k8	160	8	0.81	5	37	576.0	599	3.99%	59.46	0.49
P-n60-k10	120	10	0.94	5	37	744.0	779	4.70%	64.49	8.16
P-n60-k15	80	15	0.94	5	37	968.0	986	1.86%	64.08	21.68
P-n65-k10	130	10	0.94	5	37	792.0	811	2.40%	41.46	2.77
P-n70-k10	135	10	0.97	5	37	827.0	838	1.33%	44.72	16.55
P-n76-k4	350	4	0.97	1	37	593.0	622	4.89%	48.15	1.08
P-n76-k5	280	5	0.97	1	37	627.0	635	1.28%	48.56	9.61

Table 21: NOFSS solution cost vs. HGS solution cost on the set P.