THE UNIVERSITY OF ADELAIDE

MASTER'S THESIS

Deep Learning for Bipartite Assignment Problems

Author:
Daniel Gibbons

Supervisor:
Dr. Cheng-Chew Lim
Dr. Peng Shi

A thesis submitted in fulfillment of the requirements for the degree of Master of Philosophy

in the

School of Electrical and Electronic Engineering

August 2019

Declaration

I acknowledge the support I have received for my research through the provision of an Australian Government Research Training Program Scholarship.

Signed: .... Date: ...5/8/19...........

Statement of Authorship

Title of Paper: Deep Learning for Bipartite Assignments

Publication Status: Accepted for Publication

Publication Details: IEEE International Conference on Systems, Man and Cybernetics 2019

Principal Author

Name of Principal Author (Candidate): Daniel Gibbons

Contribution to the Paper: Thought of the core idea, wrote the code, performed experiments, carried out the analysis and wrote up the paper.

Overall Percentage: 95%

Certification: This paper reports on original research I conducted during the period of my Higher Degree by Research candidature and is not subject to any obligations or contractual agreements with a third party that would constrain its inclusion in this thesis. I am the primary author of this paper.

Signed: . ... Date: 5/8/19

Co-Author Contributions

By signing the Statement of Authorship, each author certifies that:

1. The candidate's stated contribution to the publication is accurate (as detailed above)
2. Permission is granted for the candidate in include the publication in the thesis
3. The sum of all co-author contributions is equal to 100% less the candidate's stated contribution.

Name of Co-Author: Cheng-Chew Lim

Contribution to the Paper: Supervised development of work and offered editing suggestions.

Signed: .... Date: 5/8/19

Name of Co-Author: Peng Shi

Contribution to the Paper: Supervised development of work and offered editing suggestions.

Signed: .... Date: 5 Aug 2019

Abstract

A recurring problem in autonomy is the optimal assignment of agents to tasks. Often, such assignments cannot be computed efficiently. Therefore, the existing literature tends to focus on the development of handcrafted heuristics that exploit the structure of a particular assignment problem. These heuristics can find near-optimal assignments in real-time. However, if the problem specification changes slightly, a previously derived heuristic may not longer be applicable.

Instead of manually deriving a heuristic for each assignment problem, this thesis considers a deep learning approach. Given a problem description, deep learning can be used to find near-optimal heuristics with minimal human input. The main contribution of this thesis is a deep learning architecture called Deep Bipartite Assignments (DBA), which can automatically learn heuristics for a large class of assignment problems. The effectiveness of DBA is demonstrated on two NP-Hard problems: the weapon-target assignment problem and the multi-resource generalised assignment problem. Without any expert domain knowledge, DBA is competitive with strong, handcrafted baselines.

Contents

List of Abbreviations

A-*L	**A**ttention-based architecture
A2C	**A**dvantage **A**ctor **C**ritic
AC	**A**ctor **C**ritic
BAP	**B**ipartite and **A**ssignment **P**roblem
CDF	**C**umulative **D**istribution **F**unction
CNN	**C**onvolutional **N**eural **N**etwork
DNN	**D**eep **N**eural **N**etwork
GA	**G**enetic **A**lgorithm
GAE	**G**eneralised **A**dvantage **E**stimation
GAP	**G**eneralised **A**ssignment **P**roblem
GC	**G**reedy **C**onstruction
GH	**G**reedy **H**euristic
GNN	**G**raph **N**eural **N**etwork
KP	**K**napsack **P**roblem
LAP	**L**inear **A**ssignment **P**roblem
LSTM	**L**ong **S**hort **T**erm **M**emory
MDP	**M**arkov **D**ecision **P**rocess
MKP	**M**ultidimensional **K**napsack **P**roblem
MLP	**M**ulti **L**ayer **P**erceptron
MMR	**M**aximum **M**arginal **R**eturn
MPNN	**M**essage **P**assing **N**eural **N**etwork
MRGAP	**M**ulti-**R**esource **G**eneralised **A**ssignment **P**roblem
MSE	**M**ean **S**quared **E**rror
NLP	**N**atural **L**anguage **P**rocessing
P-*L	**P**ooling-based architecture
PMF	**P**robability **M**ass **F**unction
RNN	**R**ecurrent **N**eural **N**etwork
RL	**R**einforcement **L**earning
SC	**S**imultaneous **C**onstruction
SL	**S**upervised **L**earning
TD	**T**emporal **D**ifference
TSP	**T**ravelling **S**alesman **P**roblem
WTA	**W**eapon-**T**arget **A**ssignment
u.b.	upper bound

List of Symbols

Assignment Problems	
$\mathcal{I}$	set of problem instances
$\mathcal{X}$	a particular problem instance
$\mathcal{C}$	set of problem constraints
c	a particular constraint
i	agent index
j	task index
N	number of agents
M	number of tasks
$\mathcal{Y}$	set of feasible assignment matrices
Y	assignment matrix
J	objective function
A	agent property matrix
T	task property matrix
P	agent-task pairwise array
E	environmental context vector

Neural Networks	
F	feedforward layer
σ	differentiable nonlinearity
θ	parameters
α	learning rate
L	loss function
b	bias
w	weight

Reinforcement Learning	
$\mathcal{S}$	set of states
$\mathcal{U}$	set of actions
$\mathcal{R}$	set of rewards
$\mathcal{P}$	set of transition probabilities
s	a particular state
u	a particular action
r	a particular reward
S	state as a random variable
U	action as a random variable

R	reward as a random variable
γ	discount factor
t	discrete time index
τ	final time index
π	policy
π_θ	policy parameterised by θ
g	return
G	return as a random variable
v^π	state-value function following policy π
q^π	state-action-value function following policy π
a^π	advantage function following policy π
J	a performance measure
μ^π	stationary distribution over states following policy π
$\mathcal{T}$	a recorded transition $\langle s, u, r \rangle$
b	baseline
θ	actor parameters
ϕ	critic parameters
δ	temporal-difference residual
λ	GAE hyperparameter

Custom Architecture

h	number of heads for self-attention
n	number of feedforward operations in each stack
C	communication layer
K	keys for attention
Q	queries for attention
S	number of stacks
V	values for attention
W	number of calls to DNN when using GC to construct Y^*
X	problem instance as an array with dimensions $N \times M \times \lvert X \rvert$
β	large positive infeasibility constant
ζ	pooling scalar statistic
$\boldsymbol{\zeta}$	pooling statistic row vector
Ψ	Number of rows for input to communication layer
Ω	Number of columns for input to communication layer
$\mathbb{1}_{i,j}^{\text{infeasible}}$	infeasible agent-task pair indicator
$\mathbb{1}_{i,j}^{\text{assigned}}$	agent i assigned to task j indicator
$\mathcal{E}$	embedding operation that projects X into $N \times M \times \lvert \mathcal{E} \rvert$

Applications

c	capacity
k	resource index
K	number of resources

p	profit, kill-probability
v	target value
w	weight
$\overline{og}$	optimality gap, objective gap
$\overline{OG}$	optimality gap as a random variable
$\overline{ps}$	GA population size
α	approximation ratio
μ_c	GA crossover hyperparameter
μ_m	GA mutation hyperparameter
μ_s	GA selection hyperparameter

Miscellaneous

ρ	permutation
ψ	permutation invariant function
ω	permutation equivariant function
$f(\mathcal{X})$	some method for computing Y^* for $\mathcal{X}$
$\mathbb{E}(\cdot)$	expectation
$\mathbb{P}(\cdot)$	probability
$\mathbb{R}$	the reals
$\mathbb{Z}$	the integers
$\mathcal{O}(\cdot)$	big O, worst-case runtime
$(\cdot)'$	result after some operation
$(\cdot)^*$	an optimal quantity
$\widehat{(\cdot)}$	estimate
$[\cdot]^\top$	transpose
softmax(x)	$[e^{x_1}, e^{x_2}, \ldots, e^{x_\ell}]^\top / \sum_{k=1}^{\ell} e^{x_k}$

Placeholders

$k, \ell, x, y, z, Z, \eta, \xi$	used to represent arbitrary values (e.g. indices, dimensions etc.)

Chapter 1

Introduction

A recurring problem in autonomy is that of assigning agents to tasks. Assignment problems appear throughout domains such as logistics, robotics and defence (Öncan, 2007). The well-known linear assignment problem (LAP) can be solved in cubic time (Jonker and Volgenant, 1987). However, in general, solving assignment problems to optimality is computationally infeasible and so heuristics are often employed to find near-optimal solutions.

The development of a heuristic usually requires expert-knowledge to exploit the problem structure in some way such that near-optimal solutions can be found efficiently. However, if the problem description changes slightly, a previously derived heuristic may no longer be appropriate.

Rather than handcrafting a separate heuristic for every assignment problem, this thesis explores a general-purpose learning approach. Given a description of an assignment problem, such a learning approach automatically explores the problem description and builds a black box solver. The black box solver can then be queried for fast, near-optimal solutions to specific problem instances.

Deep neural networks (DNNs) are characterised by initially requiring significant compute, but can be queried efficiently at runtime. Over the last decade, DNNs have been used to produce state-of-the-art results across diverse domains such as computer vision (Krizhevsky et al., 2012), machine translation (Vaswani et al., 2017) and game playing (Mnih et al., 2015). More recently, DNNs have been used to automatically find heuristics for classic combinatorial optimisation problems such as the travelling salesman problem (TSP) (Bello et al., 2016).

A DNN approach requires two fundamental components: an architecture and a learning algorithm. The architecture describes how data flows from the input to the output of the DNN. As the data is processed, it interacts with the DNN's internal parameters. These parameters are tuned by the learning algorithm.

The most well-known deep learning architectures are usually unsuitable for assignment problems due to issues regarding parameter-sharing and permutation equivariance. In this work, a customised architecture called Deep Bipartite Assignments (DBA) is presented that can be applied with minimal alteration to a large class of assignment problems.

1.1 Assignment Problems

There is no universal definition for what formally constitutes an assignment problem. However, the formulation this thesis presents is general enough to capture many of the most well-known assignment problems.

Definition 1. *An assignment problem is composed of a set of problem instances $\mathcal{I}$, a set of assignment constraints $\mathcal{C}$ and an objective function J. A problem instance $\mathcal{X} \in \mathcal{I}$ describes a realisation of a particular assignment problem for N agents and M tasks indexed by i and j respectively. A mapping from agents to tasks is encapsulated by a binary-valued assignment matrix $Y \in \{0,1\}^{N \times M}$. If agent i is assigned to task j, then $Y_{i,j} = 1$. Otherwise, $Y_{i,j} = 0$. A set of additional constraints $\mathcal{C}$ may be placed on Y. Let the set of all constraint-satisfying assignment matrices be given by $\mathcal{Y} = \{Y : c(Y) \text{ is satisfied } \forall\, c \in \mathcal{C}\}$. An objective function can then be defined by $J : \mathcal{I} \times \mathcal{Y} \rightarrow \mathbb{R}$. An assignment problem instance is solved by finding an optimal assignment matrix Y^* that globally optimises J for fixed $\mathcal{X}$.*

Assignment problems can rarely be solved by exhaustive-search, even for relatively small problem instances. For example, if a particular assignment problem mandates that each agent select a single task (but a particular task may be selected by more than one agent), then there are M^N possible assignment matrices. In such a case, a problem instance with $N = 20$ agents and $M = 20$ tasks has $20^{20} \approx 10^{26}$ possible assignment matrices, which cannot be searched exhaustively in real-time. If an effective lower-bounding strategy can be derived, then branch and bound can be used to can greatly speed up an exhaustive search. However, such methods may still scale poorly with increasing N and M. There may also be many assignment problems that do not have an obvious lower bounding strategy.

This thesis is especially interested in difficult assignment problems that have the following qualities:

- No practical lower bounding strategy - which prevents the application of branch and bound.
- A computationally expensive objective function J - which prohibits the use of a random search method such as a genetic algorithm (GA).
- Problem instances that require high dimensional representation - so that it is difficult to manually derive a good heuristic.

Many assignment problems are *at-least* NP-Complete (for example, the quadratic assignment problem, the weapon-target assignment problem, the generalised assignment problem etc.). Therefore, it is unrealistic to expect that optimal solutions can be found for large problem instances in real-time. Instead, near-optimal solutions can be accepted if they can be found efficiently. The quality of a solution method is a weighted combination of how long it takes to find sub-optimal solutions, and how far away from optimal the proposed solutions are. The trade-off between efficiency and optimality is generally a matter of user-preference and usually depends on the end-application.

1.2 Bipartite Assignment Problems

A deep learning approach should not be limited to one particular assignment problem. In principle, deep learning is a general-purpose paradigm that can easily be adapted to new problems. However, it is unrealistic to design a DNN that can handle any assignment problem according to the extremely general definition given in Definition 1. Therefore, this thesis considers a particular class of assignment problems called bipartite assignment problems (BAPs).

A BAP is an assignment problem that can be represented by a bipartite multigraph. A bipartite graph is a graph where every edge connects vertices from two disjoint sets. This thesis imagines that the two disjoint sets are the set of agents, and the set of tasks. Each edge provides useful information (in the form of a scalar) about a particular agent-task pair. The term multigraph implies that more than one edge can exist between two vertices. In general, this thesis assumes that the bipartite multigraph is complete, meaning every valid vertex combination (i.e. every agent-task combination) has the same number of edges. See Figure 1.1 for a visual depiction of a BAP.

The BAP definition also allows for information to be stored at each vertex of the bipartite multigraph. Such information can be used to describes agent-level or task-level properties. Finally, BAPs allow for the inclusion of a global state that is shared across all agents and tasks. Such a global state may affect the objective and so should be taken into account when computing the assignment matrix.

A large number of assignment problems are BAPs (e.g. the linear assignment problem, the weapon-target assignment problem). Note that there are many assignment problems that cannot be represented on bipartite multigraphs (most notably, the quadratic assignment problem, which requires a directed edge between every agent-agent pair). However, it is of the author's opinion that BAPs are general enough to allow for many interesting custom assignment problems.

The following is a formal definition of BAPs, in a form that is more amenable for deep learning.

Definition 2. *A bipartite assignment problem is an assignment problem with problem instances that can be represented by the tuple $\mathcal{X} = \langle A, T, P, E \rangle$, where,*

- $A \in \mathbb{R}^{N\times|A|}$*, is an agent property matrix, where each row* $A_i \in \mathbb{R}^{|A|}$ *is a vector of information specific to each agent.*
- $T \in \mathbb{R}^{M\times|T|}$*, is a task property matrix, where each row* $T_j \in \mathbb{R}^{|T|}$ *is a vector of information specific to each task.*
- $P \in \mathbb{R}^{N\times M\times|P|}$*, is an agent-task pairwise three-dimensional array, where the element* $P_{i,j} \in \mathbb{R}^{|P|}$ *is a vector that describes how agent i interacts with task j.*
- $E \in \mathbb{R}^{|E|}$*, is an environmental context vector that contains any additional information that is shared across all agents and tasks.*

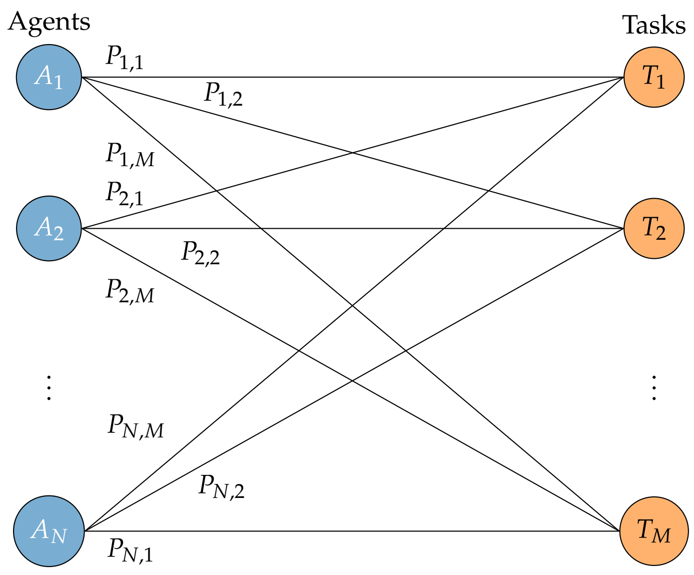


FIGURE 1.1: Visual representation of a bipartite assignment problem. Each edge contains information corresponding to a particular agent-task pair. Each vertex also contains its own information. In addition, there may also be global state information E shared across all agents and tasks (not depicted). The A_i, T_j and $P_{i,j}$ are vectors of length $|A|$, $|T|$ and $|P|$ respectively.

Example: The Linear Assignment Problem

The linear assignment problem (LAP) is often simply referred to as "the assignment problem" and is well known in the combinatorial optimisation literature. Each agent-task pairing has an associated profit $p_{i,j}$. LAP seeks the maximisation of,

$$J_{\text{LAP}} = \sum_i \sum_j p_{i,j} Y_{i,j}, \tag{1.1}$$

subject to

$$\sum_j Y_{i,j} = 1, \text{ for all } i \text{ if } N \leq M, \tag{1.2}$$

or

$$\sum_i Y_{i,j} = 1, \text{ for all } j \text{ if } N > M. \tag{1.3}$$

where the constraints specify that each agent must select a single task (unless there are more agents than task, in which case, each task must be selected by a single agent).

It is well known that LAPs can be solved in polynomial time. Assuming $N = M$, the original Hungarian algorithm gives exact solutions in $\mathcal{O}(N^4)$ (Kuhn, 1955). Later variants such as the JV algorithm bring this complexity down to $\mathcal{O}(N^3)$ (Jonker and Volgenant, 1987).

The LAP is a BAP as it can be represented by the tuple $\mathcal{X} = \langle A, T, P, E \rangle$, where,

- A is unused ($|A| = 0$).
- T is unused ($|T| = 0$).

- P is reduced to a matrix of agent-task profits ($|P| = 1$). That is, $P_{i,j} = p_{i,j}$.
- E is unused ($|E| = 0$).

1.3 Summary of Original Contributions

The main contributions of this thesis are as follows:

- A novel deep learning architecture called Deep Bipartite Assignments (DBA) that has been specifically designed for representing and understanding a large class of practical assignment problems. DBA produces high quality assignments in polynomial time with minimal human knowledge about the assignment problem itself. DBA is modular and extendable - and so easily allows for future innovations in the field of deep learning to be integrated.
- A thorough review of well-known deep learning architectures and their applicability to assignment problems.
- To the best of the author's knowledge, this thesis is the first time that techniques from the modern deep learning movement have been applied to both the weapon-target assignment (WTA) problem and the multi-resource generalised assignment problem (MRGAP). In both cases, DBA is competitive with strong handcrafted baselines.

1.4 Thesis Structure

The core content of this thesis has been accepted for publication as part of the IEEE International Conference on Systems, Man and Cybernetics 2019. This thesis significantly expands upon the conference paper to include a thorough technical background, a broad literature review, a more in-depth presentation of DBA and additional experimental validation. Where the original conference paper only considers the WTA problem, here, this thesis also considers the MRGAP and shows that DBA is general enough to still be competitive with state-of-the-art heuristics that have been handcrafted by human experts.

This thesis is organised as follows:

- Technical Background: This thesis begins with all of the necessary deep learning background required to understand and implement the thesis contributions. This section starts with a general introduction to deep neural networks and then present key results from the reinforcement learning literature.
- Design Considerations: In this short chapter, the need for a novel neural architecture is presented. Fundamental issues with conventional DNNs are raised and specifications are given for a DNN that is suitable for representing and solving BAPs.
- Literature Review: Conventional DNNs are unsuitable for representing the graph-like structure of BAPs. The deep learning literature is explored to learn how other authors have addressed similar issues of representation. The design of DBA is informed by works

from disparate areas such as natural language processing, multi-agent control and combinatorial optimisation.

- A Deep Learning Architecture for Bipartite Assignments: A novel DNN architecture entitled Deep Bipartite Assignments (DBA) is presented that can be used to represent any BAP. The architecture is presented as a series of modules. These modules can be customised and improved as future innovations from the deep learning community emerge.
- Applications: DBA is applied to two NP-Hard BAPs: the WTA problem, and the MR-GAP. Strong numerical results are provided, and there is discussion of DBA's runtime and training dynamics.
- Conclusions: This thesis finishes with a thorough discussion on the successes and limitations of DBA. Possible directions for future research are provided.

Chapter 2

Technical Background

This chapter contains all of the necessary technical background for this thesis. The techniques described in this chapter are well known in the deep learning literature and should not be considered as original contributions.

A deep neural network (DNN) takes a multi-dimensional input and processes it through a composition of differentiable layers parameterised by θ. The most well-known layer is the feedforward layer F, which is composed of an affine mapping followed by a differentiable non-linearity. Assuming vector input x,

$$F(x;\theta_k) = \sigma(\theta_k^w x + \theta_k^b)\,, \tag{2.1}$$

where θ_k^w is a weight matrix for layer k, θ_k^b is a bias vector for layer k and $\sigma(\cdot)$ is an elementwise differentiable activation function such as $\tanh(\cdot)$. As each layer is simply an affine mapping followed by a differentiable function, the output of the DNN is differentiable with respect to all of its internal parameters θ. If an appropriate loss function L is supplied, the DNN's parameters can be iteratively improved using the gradient descent equation

$$\theta' = \theta - \alpha \nabla_\theta L\,, \tag{2.2}$$

where $\alpha \in \mathbb{R}_{>0}$ is the learning rate.

The feedforward layer is just one of the many common layers employed in modern deep learning architectures. Other popular layers include the convolutional layer (Krizhevsky et al., 2012) and the long short-term memory (LSTM) cell (Hochreiter and Schmidhuber, 1997). These layers share the differentiable properties of the feedforward layer, and so their internal parameters can also be improved with gradient descent.

DNNs are popular for a number of reasons. They can theoretically approximate any function to arbitrary precision (Hornik et al., 1990). As DNNs become wider (i.e. the weight matrices become larger) and deeper (i.e. more layers are stacked on top of each other), they are more likely to find local minima with approximately equivalent performance to global minima (Choromanska et al., 2015). With modern deep learning libraries, sophisticated DNNs can be designed. Researchers can rely on autodifferentiation software to automatically compute partial derivatives for such DNNs (Abadi et al., 2016). Finally, DNNs are fast to query at test-time as they are composed of relatively simple matrix multiplications. Training and querying DNNs has become even faster in recent years with the rise of GPU technology (Raina et al., 2009).

There are a number of ways to train DNNs. This thesis consider two families of learning algorithms: supervised learning (SL) and reinforcement learning (RL).

2.1 Supervised Learning

SL provides a methodology for learning a mapping from input to output using a dataset of desirable input-output pairs. In the context of bipartite assignment problems (BAPs), there may be a dataset of optimal $\langle \mathcal{X}, Y^* \rangle$ pairs. From this dataset, SL can be used to learn a function that maps from any $\mathcal{X} \in \mathcal{I}$ to the corresponding optimal assignment matrix Y^*.

Let $\widehat{y} \in \mathbb{R}^\eta$ be the output from a DNN parameterised by θ. In SL, the loss function is typically an expectation of the difference between the DNN outputs $\widehat{y}$ and the true output y as described by the training dataset. In the case of regression, an example loss function may be the sum of ℓ_2 norms over a subset of the training dataset,

$$L = \sum_{k=1}^{\xi} \sum_{z=1}^{\eta} \left(\widehat{y_z^k} - y_z^k \right)^2 , \tag{2.3}$$

where k is used to index ξ training examples, and $z \in \{1, 2, \ldots, \eta\}$ is used to index the dimension of y. In the case of discrete classification, the cross-entropy loss is often used,

$$L = \sum_{k=1}^{\xi} \sum_{z=1}^{\eta} y_z^k \log \left(\widehat{y_z^k} \right) , \tag{2.4}$$

where it is assumed $y_z^k \in \{0, 1\}$ and $\sum_z y_z^k = 1$, where y_z^k is equal to unity if, for training example k, the correct class is class z. It should be noted that there are many commonly used SL loss functions (e.g. hinge, Huber, sum of ℓ_1 norms etc.). These loss functions are all differentiable with respect to the output of the DNN, and so can all be minimised by gradient descent.[1]

Training DNNs by SL tends to be relatively stable with modern deep learning architectures and techniques. The main limitation of SL is that is requires a training dataset of correct input-output examples. Creating such a dataset is infeasible for many applications, especially if the desired output is unknown. The other training method which considered in this thesis, RL, is, by comparison, fickle and unstable. Therefore, in this thesis, SL is used in the first instance to verify that the architecture Deep Bipartite Assignments (DBA) is actually suitable for solving BAPs. RL is then used to show that DBA can learn to solve BAPs without having access to a set of optimal training examples.

2.2 Reinforcement Learning

Consider an agent that can take actions to transition between states of an environment. Assume that the agent receives a user-defined numerical reward for every state-to-state transition. The agent's goal is to take actions that maximise the amount of reward it receives over time. Such a

[1]Not all SL loss functions are strictly differentiable throughout their domains. However, they are "differentiable enough" and work well in practical settings.

problem is typically called a Markov Decision Process (MDP). The field of reinforcement learning (RL) introduces iterative, general-purpose algorithms for solving MDPs (Sutton and Barto, 2018).

A classical MDP consists of a finite set of states $\mathcal{S}$, a finite set of actions $\mathcal{U}$, a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{U} \times \mathcal{S} \rightarrow \mathbb{R}$, transition probabilities $\mathcal{P} : \mathcal{S} \times \mathcal{U} \times \mathcal{S} \rightarrow [0, 1]$ and an optional discount factor $\gamma \in [0, 1]$. An MDP is a discrete-time process with the following event loop. At time-step t, the agent observes its state $s_t \in \mathcal{S}$. The agent then consults a (usually stochastic) policy $\pi(u_t|s_t)$ which returns a probability distribution over possible actions. The agent samples an action $u_t \sim \pi(u_t|s_t)$ and transitions to a new state s_{t+1} with probability $\mathbb{P}(s_{t+1}|s_t, u_t)$ as described by $\mathcal{P}$. Upon transitioning to state s_{t+1}, the agent receives a reward $r_t(s_t, u_t, s_{t+1})$ according to $\mathcal{R}$. This process repeats either indefinitely or until the agent reaches a terminal state.

This thesis follows the convention of Sutton and Barto, 2018 and uses upper case S, U and R to represent the states, actions and rewards as random variables. The following discussion of RL is limited to finite episodic scenarios.[2]

The following definitions will be useful throughout this thesis.

Definition 3. *The return g_t is the sum of discounted rewards experienced by the agent from time-step t until the end of the episode at time-step τ. Formally,*

$$g_t = \sum_{k=0}^{\tau - t} \gamma^k r_{t+k} \,. \tag{2.5}$$

Definition 4. *The value function $v^\pi(s)$ is the expected return from state s assuming the agent follows policy π. Formally,*

$$v^\pi(s) = \mathbb{E}_\pi \left[G_t | S_t = s \right] \,. \tag{2.6}$$

Definition 5. *The action-value function $q^\pi(s, u)$ is the expected return from state s assuming the agent follows policy π after first taking action u. Formally,*

$$q^\pi(s, u) = \mathbb{E}_\pi \left[G_t | S_t = s, U_t = u \right] \,. \tag{2.7}$$

Definition 6. *The advantage function*

$$a^\pi(s, u) = q^\pi(s, u) - v^\pi(s) \tag{2.8}$$

measures the expected difference in return by taking action u from state s (and following π thereafter) as opposed to simply following policy π from state s.

In RL, the objective is to find the optimal policy π^* that maximises the expected return over some distribution of all possible starting states. If there are a relatively small number of states, actions, and a known set of transition probabilities, algorithms from dynamic programming are guaranteed to find the optimal policy π^* (Howard, 1960). However, such methods scale poorly as the number of states and actions increase. This thesis is especially interested in finding fast, high-quality solutions where dynamic programming is too slow for real-time application.

[2]However, all of the upcoming results can be translated directly to environments with infinite time horizons.

2.2.1 The Policy Gradient

If an MDP is composed of a relatively small number of states, then the policy π can be represented by a lookup table that returns a probability mass function (PMF) over actions for every possible state. In cases where there are many (or even, an infinite number of states), a parameterised policy π_θ is often employed. A parameterised policy is simply a user-defined function that is dependant upon a number of tunable parameters θ. The only strict requirement is that the output of the parameterised policy $\pi_\theta(u|s)$ must be differentiable with respect to the policy's parameters θ. In the modern RL literature, the two most commonly used parameterised policies are linear combinations of features and deep neural networks (DNNs).[3]

Consider an arbitrary parameterised policy π_θ. The performance measure $J(\pi_\theta)$ is defined as the expected return from an arbitrary fixed starting state s_0 following policy π_θ,

$$J(\pi_\theta) = v^{\pi_\theta}(s_0)\,. \tag{2.9}$$

J is to be maximised with respect to θ. One obvious idea is to use gradient ascent to find a local maximum. That is, if the gradient in the direction of the performance measure with respect to the policy parameters $\nabla_\theta J(\pi_\theta)$ can be computed, then the parameters can be improved using

$$\theta' = \theta + \alpha \nabla_\theta J(\pi_\theta)\,, \tag{2.10}$$

where α is a small positive constant called the *learning rate*. The quantity $\nabla_\theta J(\pi_\theta)$ is often referred to as the *policy gradient*. From continual application of the above equation, the parameterised policy eventually converges to a local maximum. The well-known policy gradient theorem (Williams, 1992) states that, in the episodic case,

$$\nabla_\theta J(\pi_\theta) \propto \sum_s \mu^{\pi_\theta}(s) \sum_u q^{\pi_\theta}(s,u) \nabla \pi_\theta(u|s)\,, \tag{2.11}$$

where $\mu^{\pi_\theta}(s) \in [0,1]$ is the stationary distribution over states invoked by following policy π_θ. For a simple proof of the policy gradient theorem, see Chapter 13.2 from Sutton and Barto, 2018. From the policy gradient theorem, $\nabla_\theta J(\pi_\theta)$ can be rewritten as an expectation:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{S\sim\pi_\theta}\left[\sum_u q^{\pi_\theta}(S,u)\nabla\pi_\theta(u|S)\right]\,. \tag{2.12}$$

With some simple manipulations, the above expectation can be rewritten as

$$\begin{aligned} \nabla_\theta J(\pi_\theta) &= \mathbb{E}_{S\sim\pi_\theta}\left[\sum_u \pi_\theta(u|S) q^{\pi_\theta}(S,u)\frac{\nabla\pi_\theta(u|S)}{\pi_\theta(u|S)}\right] \\ &= \mathbb{E}_{\substack{S\sim\pi_\theta \\ U\sim\pi_\theta}}\left[q^{\pi_\theta}(S,U)\frac{\nabla\pi_\theta(U|S)}{\pi_\theta(U|S)}\right] \\ &= \mathbb{E}_{\substack{S\sim\pi_\theta \\ U\sim\pi_\theta}}\left[q^{\pi_\theta}(S,U)\nabla\log\pi_\theta(U|S)\right] \end{aligned} \tag{2.13}$$

[3]Note, a linear combination of features is the special case of a DNN with a single feedforward layer and identity activation function $\sigma(x) = x$.

The state-value function $q^{\pi_\theta}(S, U)$ is the expected return $\mathbb{E}[G]$ from being in state S, taking action U, and following π thereafter. Therefore,

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\substack{S\sim\pi_\theta \\ U\sim\pi_\theta}} \left[G \nabla \log \pi_\theta(U|S)\right] . \tag{2.14}$$

A practical algorithm is as follows (usually attributed to Williams, 1992). Over a number of episodes, collect many state-action-reward tuples $\langle s, u, r\rangle$. From this information, the returns g can be computed for each state-action pair. The returns can be used as Monte-Carlo estimates of the state-action value function q^{π_θ}. Let $\mathcal{T} = \langle s, u, g\rangle$ be a recorded transition. Assume $|\mathcal{T}|$ transitions are collected. Then, the policy gradient can be estimated by

$$\nabla_\theta J(\pi_\theta) \approx \widehat{\nabla_\theta J(\pi_\theta)} = \frac{1}{|\mathcal{T}|} \sum_{\mathcal{T}} g \nabla \log \pi_\theta(u|s) , \tag{2.15}$$

and so, the policy can be incrementally improved by stochastic gradient ascent. The above approximation is unbiased. Therefore, as the number of recorded transitions grows to infinity $|\mathcal{T}| \to \infty$, the estimation approaches the true policy gradient $\widehat{\nabla_\theta J(\pi_\theta)} \to \nabla_\theta J(\pi_\theta)$.

2.2.2 Advantage Actor-Critic

The returns-based policy gradient approximation from 2.2.1 is not typically used in practice as it is known to exhibit extremely high variance. The following equation is a well-known generalisation of the policy gradient theorem (as found in Chapter 13.3 of Sutton and Barto, 2018, for example):

$$\nabla_\theta J(\pi_\theta) = \sum_s \mu^{\pi_\theta}(s) \sum_u \left(q^{\pi_\theta}(s, u) - b(s)\right) \nabla \pi_\theta(u|s) , \tag{2.16}$$

where $b(s)$ is any function of s (normally called the *baseline*). The above expression is equivalent to the original policy gradient theorem as

$$\sum_u b(s) \nabla \pi_\theta(u|s) = b(s) \sum_u \nabla \pi_\theta(u|s) = b(s) \nabla 1 = 0 \tag{2.17}$$

Since the new expression is equivalent to the original policy gradient theorem, this new expression is unbiased. However, the user-defined function $b(s)$ can chosen such that estimates of the policy gradient have lower variance. A common choice for $b(s)$ is the state-value function $v^{\pi_\theta}(s)$. This substitution yields

$$\begin{aligned} \nabla_\theta J(\pi_\theta) &= \sum_s \mu^{\pi_\theta}(s) \sum_u \left(q^{\pi_\theta}(s, u) - v^{\pi_\theta}(s)\right) \nabla \pi_\theta(u|s) \\ &= \sum_s \mu^{\pi_\theta}(s) \sum_u a^{\pi_\theta}(s, u) \nabla \pi_\theta(u|s) . \end{aligned} \tag{2.18}$$

And so, as an expectation,

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\substack{S\sim\pi_\theta \\ U\sim\pi_\theta}} \left[a^{\pi_\theta}(S, U) \nabla \log \pi_\theta(U|S)\right] \tag{2.19}$$

If the advantage function can be computed accurately, then the policy gradient can be estimated using a similar procedure to previously. Over a number of episodes, collect many state-action-reward tuples $\langle s, u, r \rangle$. From this information, compute the advantages $a = a^{\pi_\theta}(s, u)$ for every state-action pair. Let $\mathcal{T} = \langle s, u, a \rangle$ be a recorded transition. Assume $|\mathcal{T}|$ transitions are collected. Then, the policy gradient can be estimated by

$$\nabla_\theta J(\pi_\theta) \approx \widehat{\nabla_\theta J(\pi_\theta)} = \frac{1}{|\mathcal{T}|} \sum_{\mathcal{T}} a \nabla \log \pi_\theta(u|s)\,, \tag{2.20}$$

Estimations of the above form are still unbiased but exhibit much lower variance than the returns-based approach given previously in 2.2.1. For a rigorous theoretical analysis of why this is the case, see Greensmith et al., 2004.

As the above estimation exhibits lower variance than the returns-based approach, far fewer transitions are required to accurately determine the policy gradient (that is, $|\mathcal{T}|$ can be much smaller while $\widehat{\nabla_\theta J(\pi_\theta)} \approx \nabla_\theta J(\pi_\theta)$). This in turn means that fewer iterations are required to find high-quality policies.

In practice, the advantage function is not known ahead of time. Instead, the advantage function is estimated using a *critic* parameterised by ϕ. Let $a_\phi^{\pi_\theta}(s, u)$ be the advantage of taking action u in state s and then following policy π_θ thereafter, approximated using critic parameters ϕ. The use of a critic gives rise to a family of algorithms called *actor-critic* (AC) algorithms. AC algorithms make up a large number of the current state-of-the-art RL algorithms. The RL algorithm used by this thesis uses the critic specifically for estimating advantages, and so is often referred to as an advantage actor-critic (A2C) algorithm.

2.2.3 Generalised Advantage Estimation

There are many possible procedures for estimating the advantage function. This thesis employs a popular technique called *generalised advantage estimation* (GAE) from Schulman et al., 2016.

Let v^π be the exact state-value function. One estimate of the advantage is to use the following equation

$$a^\pi(s_t, u_t) \approx \widehat{a^\pi}(s_t, u_t)_{(1)} = r_t + \gamma v^\pi(s_{t+1}) - v^\pi(s_t) \tag{2.21}$$

where the (1) in the subscript of $\widehat{a^\pi}(s_t, u_t)_{(1)}$ denotes that this is a so-called one step-estimate of $a^\pi(s_t, u_t)$. Let δ_t be the *temporal-difference* (TD) residual

$$\delta_t = r_t + \gamma v^\pi(s_{t+1}) - v^\pi(s_t)\,. \tag{2.22}$$

Hence, $\widehat{a^\pi}(s_t, u_t)_{(1)} = \delta_t$. It has not yet been stated how to compute the state-value function v^π required to yield the TD residual δ_t. This is where the aforementioned critic is used. During learning, the critic is trained to estimate $v_\phi^{\pi_\theta}(s) \approx v^\pi(s)$. Initially, the critic will give a poor approximation of the state-value function. Therefore, using a one-step advantage approximation is unlikely to be accurate. To reduce bias, a two-step estimate can be constructed using

$$\widehat{a^\pi}(s_t, u_t)_{(2)} = r_t + \gamma \left(r_{t+1} + \gamma v^\pi(s_{t+2})\right) - v^\pi(s_t)\,. \tag{2.23}$$

The two-step estimate reduces bias (as more "real" data is used) but has more variance (as the environment and policy are likely stochastic and we are only observing a single path) (Kearns and Singh, 2000). By adding zero to the right hand side, notice that

$$\begin{aligned}\widehat{a^\pi}(s_t,u_t)_{(2)} &= r_t + \gamma\left(r_{t+1} + \gamma v^\pi(s_{t+2})\right) - V(s_t) + \gamma v^\pi(s_{t+1}) - \gamma v^\pi(s_{t+1}) \\ &= r_t + \gamma v^\pi(s_{t+1}) - v^\pi(s_t) + \gamma\left(r_{t+1} + \gamma v^\pi(s_{t+2}) - v^\pi(s_{t+1})\right) \\ &= \delta t + \gamma\delta_{t+1}\,,\end{aligned} \tag{2.24}$$

and so,

$$\begin{aligned}\widehat{a^\pi}(s_t,u_t)_{(1)} &= \delta_t \\ \widehat{a^\pi}(s_t,u_t)_{(2)} &= \delta_t + \gamma\delta_{t+1}\end{aligned} \tag{2.25}$$

Using a similar process, a k-step advantage estimator can be constructed:

$$\widehat{a^\pi}(s_t,u_t)_{(k)} = \sum_{\ell=0}^{k-1} \gamma^\ell \delta_{t+\ell}\,. \tag{2.26}$$

To achieve a balance between bias and variance, GAE takes an exponentially weighted sum of all k-step estimators:

$$\widehat{a^\pi}(s_t,u_t)_{\text{GAE}} = \sum_{k=1}^{\infty} \lambda^{k-1}\widehat{a^\pi}(s_t,u_t)_{(k)}\,, \tag{2.27}$$

where $\lambda \in [0,1]$ is a user-defined parameter. An efficient procedure can now be defined for computing $\widehat{a^\pi}(s_t,u_t)_{\text{GAE}}$. Consider the following manipulations:

$$\begin{aligned}\widehat{a^\pi}(s_t,u_t)_{\text{GAE}} &= \widehat{a^\pi}(s_t,u_t)_{(1)} + \lambda\widehat{a^\pi}(s_t,u_t)_{(2)} + \lambda^2\widehat{a^\pi}(s_t,u_t)_{(3)} + \dots \\ &= \delta_t + \lambda(\delta_t + \gamma\delta_{t+1}) + \lambda^2(\delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+1}) + \dots \\ &= (1 + \lambda + \lambda^2 + \dots)\delta_t + (\lambda + \lambda^2 + \lambda^3 \dots)\gamma\delta_{t+1} + (\lambda^2 + \lambda^3 + \lambda^4 + \dots)\gamma^2\delta_{t+2} + \dots \\ &= \frac{1}{1-\lambda}\delta_t + \frac{\lambda}{1-\lambda}\gamma\delta_{t+1} + \frac{\lambda^2}{1-\lambda}\gamma^2\delta_{t+2} + \dots \\ &= \frac{1}{1-\lambda}\left(\delta_t + \gamma\lambda\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \dots\right)\end{aligned} \tag{2.28}$$

The above expression can be multiplied by the constant factor $(1-\lambda)$ to give the generalised advantage estimator as a sum of exponentially weighted TD residuals:

$$\widehat{a^\pi}(s_t,u_t)_{\text{GAE}} = \sum_{\ell=0}^{\infty} (\gamma\lambda)^\ell\,\delta_{t+\ell}\,.^4 \tag{2.29}$$

For a more detailed discussion of GAE, see Schulman et al., 2016. However, the original paper leaves off a useful recurrence relation that is required for actual implementation:

[4] This is legitimate in the context of the policy gradient equation where direction in parameter space is the fundamental consideration. Gradient direction is invariant to multiplication by a positive scalar.

$$
\begin{aligned}
\widehat{a^{\pi}}(s_t, u_t)_{\text{GAE}} &= \sum_{\ell=0}^{\infty} (\gamma\lambda)^{\ell}\, \delta_{t+\ell}. \\
&= \delta_t + \sum_{\ell=0}^{\infty} (\gamma\lambda)^{\ell}\, \delta_{t+\ell} \\
&= \delta_t + \gamma\lambda \sum_{\ell=1}^{\infty} (\gamma\lambda)^{\ell-1}\, \delta_{t+\ell} \\
&= \delta_t + \gamma\lambda \sum_{\ell=0}^{\infty} (\gamma\lambda)^{\ell}\, \delta_{t+\ell+1} \\
&= \delta_t + \gamma\lambda \widehat{a^{\pi}}(s_{t+1}, u_{t+1})_{\text{GAE}}
\end{aligned}
\tag{2.30}
$$

2.2.4 Implementation

Here, an implementation of A2C implementation is presented. It can be assumed that this implementation is used whenever RL is mentioned in the applications chapters of this thesis.

Begin by initialising the agent (or actor) parameters θ and the critic parameters ϕ. It is always assume that the actor and the critic use the same neural architecture, but with their own learned parameters (θ and ϕ respectively).[5]

The implementation considers ϵ parallel environments (as popularised by Mnih et al., 2016). Using parallel environments decreases the amount of correlation between samples and so can help to calculate less biased estimates of the policy gradient. Another advantage to using parallel environments is that, with a single batched query to the DNN, actions and critic estimates can be computed for many environments in parallel. With modern multi-CPU and GPU systems, the amount of time required to compute actions and critic estimates is decreased by roughly a factor of ϵ.

From each environment, $|\mathcal{T}|$ transitions of information are collected, where each transition consists of

$$
\mathcal{T} = \langle s, u, r, \mathbb{1}, v_{\phi}^{\pi_\theta} \rangle\,, \tag{2.31}
$$

where $\mathbb{1}$ is an indicator variable equal to unity if state s is nonterminal and $v_{\phi}^{\pi_\theta}$ is the critic's estimate of the state-value function $v^{\pi_\theta}(s)$. From this information, two additional quantities are computed: the advantage estimation $\widehat{a^{\pi_\theta}}$ and a state-value target $\widehat{v^{\pi_\theta}}$. The advantage estimation is used in the policy gradient equation to update the agent's parameters and the state-value target is used to help guide the critic learn the true state-value function $v_{\phi}^{\pi_\theta}(s) \approx v^{\pi_\theta}(s)$ for all $s \in \mathcal{S}$. The more accurate the critic becomes, the more accurate the advantage estimates become.

It is assumed that every environment episode is isolated. Without loss of generality, assume that at time $t = 0$ the agent is in a nonterminal state (that is $\mathbb{1}_t = 1$). At time $t = \tau$, the agent has either reached a terminal state (that is $\mathbb{1}_t = 0$) or $\tau = |\mathcal{T}|$, which implies that the data collection procedure has been halted to allow for a parameter update to occur. The advantages are computed using Algorithm 1 (as taken from Dhariwal et al., 2017). Line 4 of Algorithm 1 uses the recurrence relation from the end of 2.2.3. Once the advantages have been estimated, the

[5] In some works, the actor and the critic use the same parameters until the last layer of the DNN. In others, the architectures for the actor and the critic are completely different.

Algorithm 1 Generalised advantage estimation

```
1: $\widehat{a_\tau^{\pi_\theta}} \leftarrow r_\tau + \mathbb{1}_\tau v_{\tau+1,\phi}^{\pi_\theta} - v_{\tau,\phi}^{\pi_\theta}$
2: **for** $t \in \{\tau-1, \tau-2, \dots, 1, 0\}$ **do**
3:     $\delta_t \leftarrow r_t + \gamma v_{t,\phi}^{\pi_\theta} - v_{t+1,\phi}^{\pi_\theta}$
4:     $\widehat{a_t^{\pi_\theta}} \leftarrow \delta_t + \gamma\lambda \widehat{a_{t+1}^{\pi_\theta}}$
```

state-value targets are computed simply using

$$\widehat{v_{t,\phi}^{\pi_\theta}} = \widehat{a_t^{\pi_\theta}} + v_{t,\phi}^{\pi_\theta} \tag{2.32}$$

for all $t \in \{0, 1, \dots, \tau\}$. An SL procedure with a regression loss (such as mean-squared-error (MSE) or the Huber loss) is then used to update the critic parameters ϕ such that the critic approaches the true state-value function.

Summary

This chapter gave an overview of DNNs and described learning algorithms from two contrasting methodologies: SL and RL. The bulk of this chapter was devoted to a state-of-the-art RL algorithm called advantage actor-critic (A2C) with generalised advantage estimation (GAE). The main takeaway from the RL section is that, by simply collecting states, actions, and rewards, the DNN's parameters can be continually improved to perform some desired function. For the purposes of this thesis, the states are the BAP instances $\mathcal{X} \in \mathcal{I}$, the actions are the assignment matrices $Y \in \mathcal{Y}$, and the rewards are governed by the objective $J(\mathcal{X}, Y)$. Later, in 5.4, more details will be provided describing exactly how RL is used to train DBA to solve BAPs.

Chapter 3

Design Considerations

This thesis presents a novel architecture for automatically finding heuristic solutions to bipartite assignment problems (BAPs). However, it has not yet been discussed why such an architecture is required. This short chapter examines the most commonly used approach from the deep learning literature, and explains why it is unsatisfactory for representing and solving BAPs.

Recall that a particular BAP is defined by a set of instances, a set of constraints and an objective function $\langle \mathcal{I}, \mathcal{C}, J \rangle$. A deep neural network (DNN) is to take problem instances $\mathcal{X} \in \mathcal{I}$ and return optimal assignment matrices Y^*.

A naïve first attempt to design such a DNN is to use a composition of feedforward layers. The information contained in $\mathcal{X}$ is first reduced into a one-dimensional array x of length $|x| = N|A| + M|T| + NM|P| + |E|$. This vector can then be passed through a series of feedforward layers to yield an array of length MN. Each element of the output array is a scalar corresponding to a unique agent-task combination. Depending on the problem description and the chosen training method (SL or RL), simple operations can be performed to construct a feasible assignment matrix Y. Such an approach (flattening out all of the problem information and then passing through many feedforward layers) is often referred to as a multilayer perceptron (MLP) and is commonplace throughout the deep learning literature. However, there are two fundamental issues that prevent the adaptation of a naïve MLP to BAPs: variance to agent/task permutation and parameter dependence on N and M.

Permutation Variance

The naïve MLP approach is sensitive to the ordering of the agents and tasks. That is, if the positions of two agents and/or tasks are swapped at the input, the DNN may produce a different assignment matrix. Ideally, the DNN should be invariant to the ordering of the input information. The input information should be considered as an unordered set as opposed to an ordered tuple or vector. It is therefore required that the DNN be *permutation equivariant* with respect to the ordering of both agents and tasks. Loosely speaking, this means that, if the information of two agents and/or two tasks is swapped, the DNN should output an *equivalent* assignment matrix.

Permutation equivariance is closely linked to the concept of permutation invariance. A permutation invariant function ψ has the property that $\psi(z) = \psi(\rho(z))$, where z is an ordered tuple of elements and $\rho(z)$ is an arbitrary permutation of the elements of z. A permutation equivariant function ω on the other hand requires that $\omega(\rho(z)) = \rho(\omega(z))$. That is, if the function takes a tuple permuted by ρ, the output of the function should be the same as if the function was applied to the original tuple and then permuted by ρ.

To demonstrate these concepts more concretely, consider a simple assignment problem with a single agent and M tasks. Each task has a cost $c_j \in c$. The agent is required to select the task with the least cost. Here, the minimum cost is a permutation invariant function. Regardless of how the elements of c are shuffled, $\min(c)$ is constant. The optimal assignment matrix however, is Y^* is permutation equivariant. As an example, if the cost vector is $c = [\ 3\ 1\ 6\ 2\ ]^\top$ then $Y^* = [\ 0\ 1\ 0\ 0\ ]$. If c is permuted with an arbitrary 1-index tuple, say $\langle\ 3\ 1\ 4\ 2\ \rangle$, then $\rho(c) = [\ 6\ 3\ 2\ 1\ ]^\top$. The optimal assignment matrix is now $[\ 0\ 0\ 0\ 1\ ]$, which equal to the original optimal assignment matrix permuted by ρ.

Parameter Dependence on N and M

A naïve MLP approach is restricted to a fixed number of N agents and M tasks. For example, at the input, the number of weights for the first feedforward layer F_1 is conditioned on both N and M as $\theta_1^w \in \mathbb{R}^{|x| \times |F_1|}$, where $|F_1|$ is the number of neurons in the first feedforward layer. As the weight matrix has $|x| = N|A| + M|T| + NM|P| + |E|$ rows, the number of parameters is directly tied to both N and M. There is a similar issue at the output, where the number of columns in the final weight matrix is NM. Such a DNN cannot be used on larger problem instances than those seen during training as the input and output weight matrices are undefined for larger N and M. It is not even clear that such a DNN will function as intended in cases of smaller N and M. For example, say the DNN is configured to represent N_1 agents and M_1 tasks. The DNN is then presented with a problem instance with $N < N_1$ and/or $M < M_1$. Unused elements of x can be filled with null placeholders. At best, such a DNN will perform unnecessary matrix multiplications on the null placeholders. At worst, the DNN will not perform as expected as it needs to make meaningful interpretations of the null placeholders in such a way that the necessary computations being undertaken on the real problem information are not affected.

Summary

This short chapter identified fundamental issues that necessitate the need for a new DNN architecture for solving BAPs. To adequately represent and solve BAPs, a DNN architecture must be permutation equivariant with respect to both agents and tasks. In addition, it is desirable that the number of DNN parameters does not explicitly depend on N or M.

Chapter 4

Literature Review

Assignment problems have received little attention in the deep learning literature. However, there are a number of relevant domains that face similar issues with regards to graph representation, permutation equivariance, and parameter dependence on input/output size. This chapter summarise a number of contributions from the deep learning literature that will aid in the design of a deep neural network (DNN) architecture for bipartite assignment problems (BAPs).

Assignment problems have not received significant attention from the deep learning literature. Emami et al., 2018 undertook a survey of a number of machine learning methods for performing multidimensional assignments for solving tracking problems. However, their survey focused more an a particular assignment problem (where this thesis seek sa more general approach). Milan et al., 2017 used a long short-term memory (LSTM) based approach for finding approximate solutions to specific formulations of the linear assignment problem and the quadratic assignment problem. However, their architecture is not amenable to more generic assignment problems and is dependent upon fixed input/output sizes.

One area that receives regular attention from the deep learning literature is combinatorial optimisation. A number of authors have found data-driven approaches such as deep learning to provide fast, high quality heuristic solutions for a number of well known combinatorial problems. These findings are extremely relevant as BAPs are a particular type of combinatorial optimisation problem.

Combinatorial optimisation problems typically require a mapping from a set or a sequence of objects to another set or sequence of objects. The canonical multi-layer perceptron (MLP) composed of feedforward layers is often inappropriate as it can only map from a vector of reals to another vector of reals. To overcome this issue, authors have designed custom DNN architectures that can adequately represent and solve combinatorial problems. This thesis identifies three relevant bodies of work: natural language processing, multi-agent communication and graph representation.

4.1 Natural Language Processing

This chapter begins by detailing a progression in neural architectures from the field of natural language processing (NLP). Although NLP appears somewhat unrelated to combinatorial optimisation, a number of innovations from the field of NLP are now commonplace in the deep learning combinatorial optimisation literature.

A seminal work in machine translation is the sequence-to-sequence (seq2seq) architecture from Sutskever et al., 2014. The seq2seq architecture uses two DNNs: an encoder and a decoder. Both DNNs have an architecture composed of LSTM cells (Hochreiter and Schmidhuber, 1997). The encoder reads in each word of a sentence and embeds the sentence into a hidden state. The hidden state is then passed to the decoder to recover the sentence, one word at a time, in a different language. Each generated word is fed back in to the decoder to compute the next word. Although the seq2seq architecture is designed for NLP, it presents a general approach for translating any sequence of objects to any other sequence of objects. The main restriction on seq2seq is that is assumes a fixed number of possible words. Therefore, seq2seq cannot be directly used for combinatorial optimisation problems where problem size may be variable.

Another important innovation from the field of NLP is the attention mechanism, as brought to prominence by Bahdanau et al., 2015. In contrast to seq2seq, which applies the decoder to a sentence encoded in a single hidden state vector, an attention mechanism instead applies a decoder across each individual word embedding in parallel. This way, no information is lost by trying to reduce the sentence down to a single vector, and the attention mechanism can easily establish relationships between words that are far away from one another in a sentence.

Vinyals et al., 2015 built upon the ideas from seq2seq and attention to create pointer networks (Ptr-Nets). Ptr-Nets are specifically designed for sequential combinatorial optimisation problems. Rather than generating words, the Ptr-Net uses an attention mechanism to point back to the inputs. For example, rather than translating a word from one language to another, Ptr-Net takes arbitrary words from a set and then points back to words within the same set. Vinyals et al., 2015 trained Ptr-Net using supervised learning (SL) to provide approximate solutions to well known problems such as convex hull computation and the travelling salesman problem (TSP).

Bello et al., 2016 took the Ptr-Net architecture and instead trained it by reinforcement learning (RL). The use of RL allowed the authors to train the Ptr-Net without access to a database of optimal solutions ahead of time. Rather, the parameters of the Ptr-Net were tuned in the direction of the policy gradient (using a similar method to the one presented in 2.2). The resultant model was able to outperform the original SL model by Vinyals et al., 2015. In addition, the authors suggested a number of methods to improve performance at test time. For example, rather than training a single architecture to solve the TSP, they trained 16 models in parallel. As the training process is stochastic, each model is able to find different tours. At test time, 16 tours can be found for a specific problem instance, and the best solution can be quickly verified. Later in the paper, Bello et al., 2016 also applied Ptr-Net (trained by RL) with successfully to the knapsack problem (KP). The KP is of particular interest as it is the special case of the generalised assignment problem (GAP) with a single agent (which is a BAP as shown in Chapter 7).

In a similar manner to Ptr-Net, Mirhoseini et al., 2017 combined the seq2seq architecture with an attention mechanism. However, rather than working on classic problems such as the TSP, they instead used their architecture to optimise how computational operations were allocated across CPUs and GPUs. Such an application is ideally suited to deep learning, as it is very difficult to capture the problem in a closed-form objective function (as the internal workings of a computer are complicated and it is not entirely obvious how operations should be linked across devices to achieve optimal results). Their work demonstrated that DNNs can be directly translated from abstract, deterministic combinatorial problems, to real-world, stochastic problems with little alteration.

Many problems from combinatorial optimisation (including assignment problems), require set-to-set mappings (instead of sequence-to-sequence mappings). Although it is possible to use Ptr-Nets with sets of objects (as in the knapsack application by Bello et al., 2016), Vinyals et al., 2016 found that the order in which the objects are presented to Ptr-Net can make a significant difference to solution quality (due to the sequential nature of the LSTMs found within Ptr-Net).

Up until 2017, LSTMs and similar recurrent mechanisms were considered essential in NLP. However, Vaswani et al., 2017, showed that, using attention alone, they could outperform seq2seq (the previous state-of-the-art) on machine translation tasks. Specifically, the authors used a variation on attention called self-attention (as first presented by Cheng et al., 2016). Rather than using a single external decoder that applies attention across every word, in self-attention, each word has its own decoder and interacts with all of the other words in the sentence. For purposes of this thesis, using attention without any LSTMs is an interesting concept it allows for sets to be processed in a permutation invariant/equivariant fashion. In fact, authors such as Deudon et al., 2018 and Kool et al., 2019, successfully adapted attention-only approaches to the TSP. In doing so, both authors reported performance improvements over the Ptr-Net baselines recorded by Bello et al., 2016. These attention-only approaches satisfy the design considerations of permutation equivariance and parameter dependence on problem size. However, it is not clear whether these approaches can be applied directly to BAPs, where interactions between two distinct sets of objects (the set of agents and the set of tasks) need to be considered.

4.2 Multi-Agent Communication

A core area of artificial intelligence is multi-agent systems. A number of authors have proposed DNN architectures that facilitate multi-agent communication. Crucially, these architectures ensure that the operations responsible for passing information between agents are differentiable, and so agent-to-agent communication can be iteratively improved with gradient descent.

Sukhbaatar et al., 2016 proposed a simple architecture called the communication neural net (CommNet). CommNet uses the following simple idea. For each agent, take the elementwise mean of all the other agent's hidden states.[1] Then, process the agent's own hidden state and the elementwise mean through two separate feedforward layers (without biases and no activation function). The two processed vectors are then summed to form the agent's next hidden state. This process can be performed an arbitrary number of times while still maintaining permutational equivariance. The authors demonstrate that CommNet successfully facilitates multi-agent collaboration across a number of simple multi-agent games. In parallel to the development of CommNet, Foerster et al., 2016 developed a similar architecture called differentiable inter-agent learning (DIAL) to pass messages between agents. However, DIAL can only pass messages between agents once per time-step, and the number of agents is fixed. Hoshen, 2017 proposed a

[1]The term "hidden state" is used to describe the state of the data at some point in the DNN before any inference is applied at the output. The state is "hidden" because it is not comprehensible to a human observer.

similar communication scheme to CommNet but included an attention mechanism to exchange messages as opposed to simply taking the elementwise mean. The use of an attention mechanism over the elementwise mean appears to facilitate more non-linearity in the communication exchanges.

Guttenberg et al., 2016 proposed a permutation equivariant architecture for predicting particle dynamics. At each time-step, their DNN takes the state of every particle in the system predicts how the states will change at the next time step. To do this, they introduce the concept of permutational layers. Given a set of objects, a permutational layer processes every pair of objects through the same feedforward layer in parallel. The output for each object is then the sum (or the mean) of all the processed pairs that include the object itself. Permutation layers can then be stacked together to arbitrary depth. This concept is similar to that of CommNet, but has running time $\mathcal{O}(n^2)$ as opposed to the $\mathcal{O}(n)$ of CommNet (where n is the number of objects).

Zaheer et al., 2018 formalised the structure of permutation equivariant (and invariant) DNNs. They presented invariant and equivariant architectures that can be considered as variations on CommNet. Empirically, they showed that such architectures are effective on a wide range of tasks such as anomaly detection and set expansion. In addition, they provided necessary and sufficient conditions for permutation equivariant and invariant architectures.

4.3 Graph Representation

The convolutional neural network (CNN) is perhaps the most iconic neural architecture of the modern deep learning movement (Krizhevsky et al., 2012). CNNs employ convolutional layers that are designed to extract features from image data (Yann et al., 1998). Convolutional layers have two appealing properties: they are translation invariant and their internal parameters are not dependent on the size of the image that they are convolving over. Unfortunately, convolutional layers are designed specifically for representing Euclidean space. This feature makes CNNs ideal for working with image data, but not for understanding information over non-Euclidean graphs. Adapting CNNs to non-Euclidean structures such as graphs has attracted significant attention in recent years, culminating in the field of geometric deep learning (Bronstein et al., 2017).

An early work in this area is that of Gori et al., 2005, who first proposed the concept of graph neural networks (GNNs). There are many variations on GNNs, but they tend to use the following procedure. At each node of the graph, some notion of state is stored as a vector. This state is then propagated out to the node's local neighbourhood across the outgoing edges of the node. As each node receives information from the other nodes in its neighbourhood, a permutation invariant transformation can be applied to update the node's internal state. This process can be repeated to gradually propagate information throughout a connected graph. At convergence, every node is fully aware of the information contained in the graph. Meaningful operations can then be applied to the nodes to infer useful information.

GNNs received little attention until they were revived as part of the modern deep learning movement by Li et al., 2016. Defferrard et al., 2016 considered GNNs directly as a generalisation from CNNs. Battaglia et al., 2016 adapted a GNN-like approach for modelling interactions between physical objects. Gilmer et al., 2017 distilled many of the most popular mechanisms for operating on graphs into a framework called message-passing neural networks (MPNNs). The MPNN framework considers three functions: a message function (to pass information between nodes), a vertex (or node) update function (to update each node's internal state), and a readout function (that distills the entire graph into a single feature vector). Some authors have applied MPNN-like architectures to combinatorial optimisation problems. Selsam et al., 2019 adapted MPNN to solve arbitrary propositional satisfiability problems. Dai et al., 2017 claim to have beaten Ptr-Nets on the TSP as well as a number of other well known graph problems such as the maximum-cut problem and the minimum vertex-cover problem.[2]

Summary

Many of the papers utilised similar concepts for facilitating arbitrary computations across sets of related objects. From a thorough review of the literature, two fundamental processes emerged: self-assessment and inter-object communication. First, every object assesses its own information using an MLP. All objects in the system typically share the same parameters, which allows for easier training and lower memory requirements. After assessing their own information, objects communicate with each other using some permutation invariant/equivariant function (such as the elementwise mean in CommNet or some sort of attention mechanism). These two processes can be stacked and repeated an arbitrary number of times to approximate sophisticated, nonlinear functions. Both processes are differentiable and so all parameters required for both self-assessment and inter-object communication can be learned given an appropriate loss function (typically from either SL or RL). From these findings, a neural architecture can be designed for representing and solving BAPs.

[2]Dai et al., 2017 employed RL like many of the other authors adapting DNNs to combinatorial optimisation. Unlike other works, the authors explicitly chose to use Q-learning over a policy gradient based method for its improved sample efficiency (Gu et al., 2017).

Chapter 5

A Deep Learning Architecture for Bipartite Assignment Problems

This chapter details the design of a deep neural network (DNN) architecture called Deep Bipartite Assignments (DBA) that is capable of taking bipartite assignment problem (BAP) instances and returning feasible assignment matrices. This chapter begins with a broad overview of DBA. Next, close attention is paid to the *communication layer*, a key component of DBA. Finally, learning algorithms for training DBA are presented and discussed.

5.1 Preliminaries

5.1.1 Array Conversion

DBA is to take a problem instance $\mathcal{X} = \langle A, T, P, E \rangle \in \mathcal{I}$ as input and return a valid assignment matrix $Y \in \mathcal{Y}$. DBA first converts $\mathcal{X}$ to a three-dimensional array $X \in \mathbb{R}^{N \times M \times |X|}$, where $|X| = |A| + |T| + |P| + |E| + 1$. X can be indexed by i and j to view information for a particular agent-task pair.

$$X_{i,j} = \begin{bmatrix} A_i \\ T_j \\ P_{i,j} \\ E \\ \mathbb{1}_{i,j}^{\text{infeasible}} \end{bmatrix} \tag{5.1}$$

For a given i and j, the first $|A|$ entries correspond to agent-specific properties, the next $|T|$ entries correspond to target-specific properties, the next $|P|$ entries correspond to agent-task pairwise information, the next $|E|$ entries correspond to any contextual information that is consistent across all agents and tasks and the final element $\mathbb{1}_{i,j}^{\text{infeasible}}$ is a binary indicator equal to unity if assigning agent i to task j would invalidate one of the problem constraints according to $\mathcal{C}$.

Taking the linear assignment problem (LAP) as an example, $|X| = |A| + |T| + |P| + |E| + 1 = 0 + 0 + 1 + 0 + 1 = 2$, which implies that $X \in \mathbb{R}^{N \times M \times 2}$. Indexing X by i and j returns $X_{i,j} = [p_{i,j}, \mathbb{1}_{i,j}^{\text{infeasible}}]^\top$.

5.1.2 Representation (Optional)

Throughout DBA, the data is maintained as a three-dimensional array with dimensions $N \times M \times \eta$ (where η is arbitrary and will often change throughout DBA . This array can be considered from four different perspectives (or representations). From these representations, all of the required computations to construct the assignment matrix Y can be performed.

The three-dimensional array can be considered in two fundamental ways: as a matrix-of-vectors, and as a vector-of-matrices. The matrix-of-vectors representation is useful when the same operation is to be applied to every agent-task combination in parallel. Performing such an operation yields equivalent results regardless of whether the data has dimensions $N \times M \times \eta$ or $M \times N \times \eta$. The vector-of-matrices approach is then used to perform computations across objects (either across tasks for a particular agent or across agents for a particular task).

Representation 1: $N \times M \times \eta$ matrix-of-vectors

Throughout DBA, it is generally assumed that the data flow as a three-dimensional array with dimensions $N \times M \times \eta$, where η is arbitrary. If the data are considered as a matrix-of-vectors, where each vector represents an agent-task pair, then each row of the matrix corresponds to a particular agent (from 1 to N) and each column corresponds to a particular task (from 1 to M).

$$\begin{array}{cc} & \overset{\text{tasks}}{\rightarrow} \\ \text{agents} \downarrow & \begin{bmatrix} \xi_{1,1} & \cdots & \xi_{1,M} \\ \vdots & \ddots & \vdots \\ \xi_{N,1} & \cdots & \xi_{N,M} \end{bmatrix} \end{array},$$

where $\xi_{i,j} \in \mathbb{R}^\eta$ is a vector of features for the pairing of agent i with task j.

Representation 2: $M \times N \times \eta$ matrix-of-vectors

It is sometimes useful to swap the first two axes of the data such that it has dimensions $M \times N \times \eta$. The resulting matrix of agent-task representations then has a row for each task and a column for each agent. In this thesis, this operation is referred to as a *transposition*. Applying a transposition to Representation 1 yields

$$\begin{array}{cc} & \overset{\text{agents}}{\rightarrow} \\ \text{tasks} \downarrow & \begin{bmatrix} \xi_{1,1} & \cdots & \xi_{1,N} \\ \vdots & \ddots & \vdots \\ \xi_{M,1} & \cdots & \xi_{M,N} \end{bmatrix} \end{array},$$

where $\xi_{j,i} \in \mathbb{R}^{\eta}$ is a vector of features for the pairing of task j with agent i. The data can be transposed again to revert to the original dimensions $N \times M \times \eta$.

Representation 3: $N \times M \times \eta$ **vector-of-matrices**

The data, as a three-dimensional array, can also be considered as a vector-of-matrices (as opposed to a matrix-of-vectors as described previously). As a vector-of-matrices with dimensions $N \times M \times \eta$, each element of the vector is a matrix that summarises all of the available tasks for a particular agent. Each matrix has dimensions $M \times \eta$, where each row corresponds to a particular task and each column corresponds to a particular feature of that agent-task combination.

$$\text{agents} \downarrow \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_N \end{bmatrix},$$

where $\xi_i = \text{tasks} \downarrow \overset{\text{features}}{\overset{\rightarrow}{\begin{bmatrix} \xi_{i,1,1} & \cdots & \xi_{i,1,\eta} \\ \vdots & \ddots & \vdots \\ \xi_{i,M,1} & \cdots & \xi_{i,M,\eta} \end{bmatrix}}}$ is a matrix of tasks for each agent.

Representation 4: $M \times N \times \eta$ **vector-of-matrices**

A transposition of Representation 3, yields a vector of matrices with dimensions $M \times N \times \eta$, where each element of the vector is a matrix that summarises the thoughts of every agent regarding a particular task. Each matrix has dimensions $N \times \eta$, where each row corresponds to a particular agent and each column corresponds to a particular feature of that agent-task combination.

$$\text{tasks} \downarrow \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_M \end{bmatrix},$$

where $\xi_j = \text{agents} \downarrow \overset{\text{features}}{\overset{\rightarrow}{\begin{bmatrix} \xi_{j,1,1} & \cdots & \xi_{j,1,\eta} \\ \vdots & \ddots & \vdots \\ \xi_{j,N,1} & \cdots & \xi_{j,N,\eta} \end{bmatrix}}}$ is a matrix of agents for each task.

5.2 Overview

From X, DBA performs a series of operations to construct a valid assignment matrix Y. A general overview of DBA is depicted in Figure 5.1.

5.2.1 Embedding

DBA begins by processing X through an embedding operation $\mathcal{E}$. The embedding operation is simply a feedforward layer that projects the last dimension of X into some other dimension $\mathbb{R}^{|\mathcal{E}|}$, where (typically) $|\mathcal{E}| > |X|$. In other words, feedforward layer parameterised by $\theta_{\mathcal{E}}$ is applied elementwise to every $X_{i,j}$ as follows:

$$\mathcal{E}(X) = \begin{bmatrix} F(X_{1,1};\theta_{\mathcal{E}}) & \dots & F(X_{1,M};\theta_{\mathcal{E}}) \\ \vdots & \ddots & \vdots \\ F(X_{N,1};\theta_{\mathcal{E}}) & \dots & F(X_{N,M};\theta_{\mathcal{E}}) \end{bmatrix} \tag{5.2}$$

and so $\mathcal{E}(X) \in \mathbb{R}^{N \times M \times |\mathcal{E}|}$.

5.2.2 Main Body

The main body of DBA is a composition of S *stacks*, where each stack contains many operations *stacked* together. From the output of the embedding layer, there is a vector representation in $\mathbb{R}^{|\mathcal{E}|}$ for every possible agent-task pair. It is necessary to exchange information across these vector representations to make informed decisions about which agent-task pairs should be included in the assignment matrix Y. DBA use so-called "communication layers" to facilitate the exchange of information across agent-task pairs, which are discussed at length in Section 5.3.

Each stack is created by chaining together three distinct operations:

1. Feedforward operation: Generalises the feedforward layer in the same way as the embedding layer. The input to the feedforward operation is a matrix of vectors, where each vector represents a particular agent-task pair. The feedforward operation applies the same parameterised feedforward layer to every agent-task vector in parallel.

2. Communication operation: A communication layer is a matrix-to-matrix function that has a number of special properties (as will be discussed in Section 5.3). The communication operation takes a vector of matrices, and applies the same (possibly parameterised) communication layer to every matrix in parallel.

3. Transposition operation: Swaps the first two axes of a three-dimensional array. For example, if the array has dimensions $N \times M \times \eta$ (where η denotes an arbitrary dimension length), then after transposition, the array has dimensions $M \times N \times \eta$.

Each stack is then composed of three sections: inter-task communication, inter-agent communication, and self-assessment. See Figure 5.2 for a schematic of each stack and Figure 5.4 for an expanded diagram that depicts the operations found within each section.

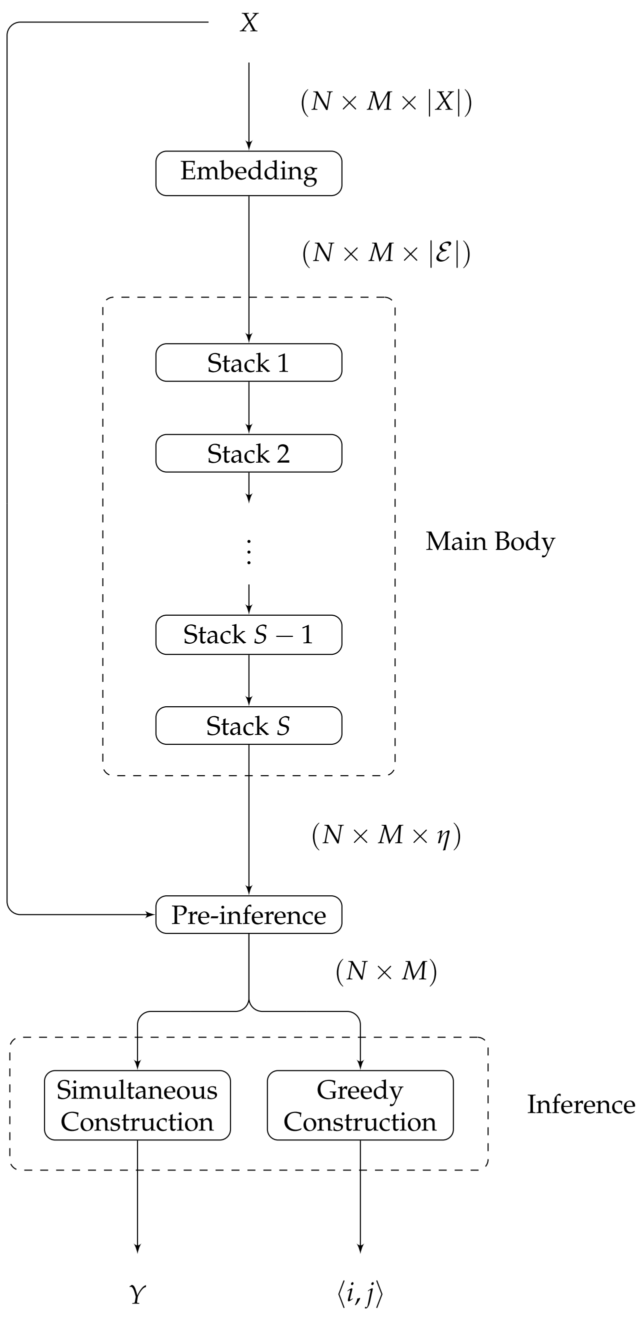


FIGURE 5.1: A general overview of DBA. Data dimensions are given in parentheses.

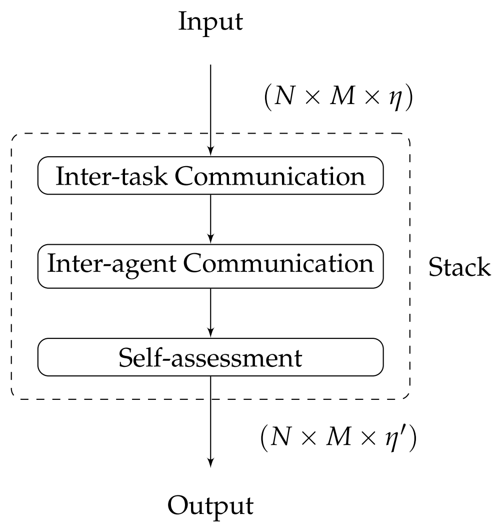

Input
$(N \times M \times \eta)$
Inter-task Communication
Inter-agent Communication
Self-assessment
Stack
$(N \times M \times \eta')$
Output

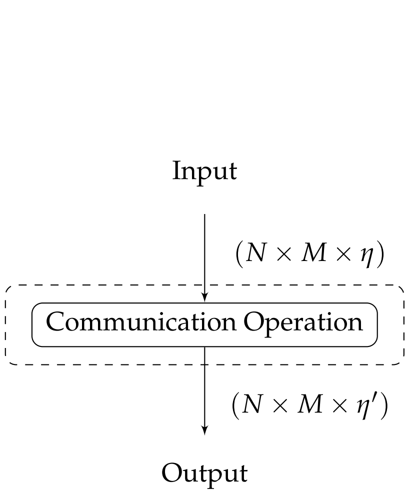
FIGURE 5.2: Stack schematic.

1. Inter-task communication: For each agent, perform computations over all available tasks. It is assumed the input has dimensions $N \times M \times \eta$ and apply a communication operation.
2. Inter-agent communication: For each task, perform computations across all agents. A transposition operation is applied to convert the array dimensions to $M \times N \times \eta$. Next, a communication operation is applied. Finally, a transposition operation is used to convert the resultant array back to dimensions $N \times M \times \eta'$ (where η' may not necessarily be equal to η as a result of the communication operation).
3. Self-assessment: After both rounds of communication, additional computation can be performed on each agent-task pair independently by applying an arbitrary number of feedforward operations in series (for all applications in this thesis, two feedforward operations are used).

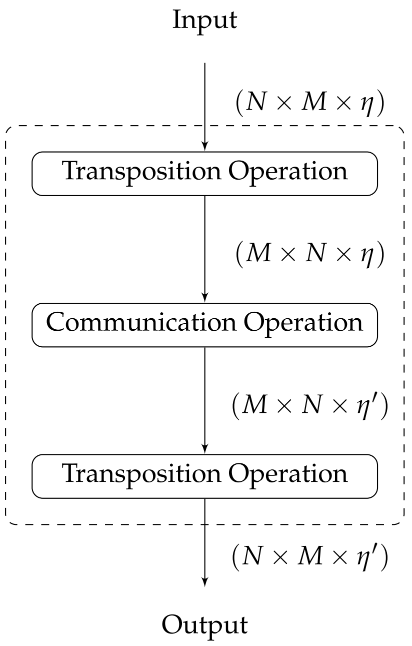
5.2.3 Pre-inference

Assume that the final stack outputs an array with dimensions $N \times M \times \eta$, where η is arbitrary. A feedforward operation is applied to map the array into $\mathbb{R}^{N \times M \times 1}$ and then the redundant final dimension is removed to yield $\widetilde{Y} \in \mathbb{R}^{N \times M}$.

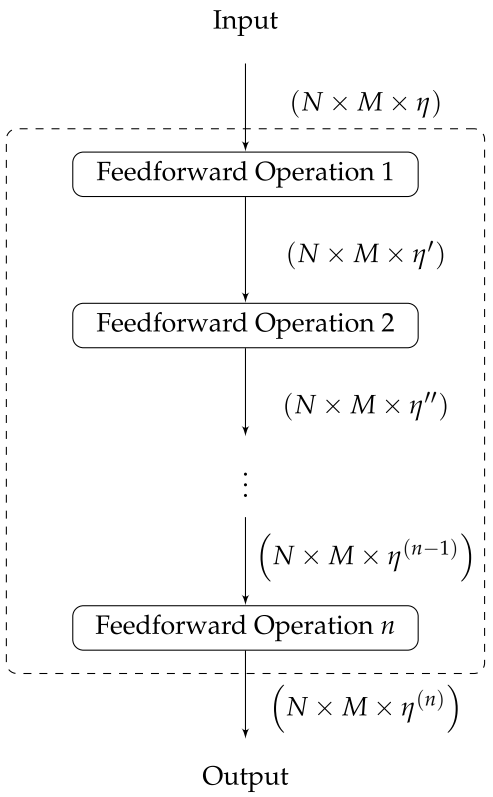
Before performing inference, infeasible agent-task pairs need to be "masked out". During inference, softmax operations are used to derive probability distributions over agent-task combinations. If infeasible agent-task pairs can be driven to large negative values, then the resulting probabilities for these agent-task pairs will be zero (assuming finite computational precision). It is then impossible for an agent-task pair with probability zero to ever be included in the final assignment matrix Y. To mask out infeasible agent-task pairs, DBA takes the temporary output,



(a) Inter-task communication.


Input
$(N \times M \times \eta)$
Transposition Operation
$(M \times N \times \eta)$
Communication Operation
$(M \times N \times \eta')$
Transposition Operation
$(N \times M \times \eta')$
Output

(b) Inter-agent communication.


Input
$(N \times M \times \eta)$
Feedforward Operation 1
$(N \times M \times \eta')$
Feedforward Operation 2
$(N \times M \times \eta'')$
$\vdots$
$\left(N \times M \times \eta^{(n-1)}\right)$
Feedforward Operation n
$\left(N \times M \times \eta^{(n)}\right)$
Output

(c) Self-assessment. This thesis always assumes $n = 2$.

FIGURE 5.3: Each section of the stack in detail.

$\widetilde{Y} \in \mathbb{R}^{N \times M}$ and applies the operation

$$\overline{Y}_{i,j} = \widetilde{Y}_{i,j} - \beta \mathbb{1}_{i,j}^{\text{infeasible}} \tag{5.3}$$

elementwise to every $\langle i, j \rangle$ pair , where β is a large positive constant (this thesis uses 2^{30}).

5.2.4 Inference

The are two methods for constructing a valid assignment matrix $Y \in \mathcal{Y}$ from $\overline{Y}$: simultaneous construction and greedy construction.

Simultaneous Construction

In the special case that each agent must choose a single task (but each task can be selected by an arbitrary number of agents), simultaneous construction (SC) can be used. SC first applies a softmax operation to each row of $\overline{Y}$ to yield a probability mass function (PMF) over tasks for each agent. An assignment matrix can then be constructed simultaneously across all agents using one of two methods:

1. Deterministic: For each agent, select whichever task has the highest value given by the agent's PMF.
2. Stochastic: Sample a task according to each agent's PMF over tasks.

The deterministic method is usually preferable for use at test-time. However, when training by reinforcement learning (RL) (specifically when using a policy-gradient based approach as in 2.2.1), the stochastic method is required to estimate the direction of the policy gradient.

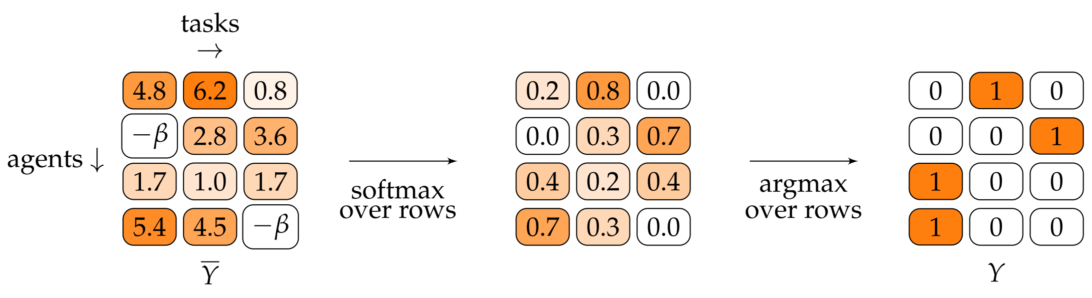
It is clear that, if each agent is to select a single task (and each task can be selected by an arbitrary number of agents), that using simultaneous construction will always give a feasible assignment matrix.[1] However, if the problem constraints are more sophisticated, the above approach may result in an infeasible assignment matrix (e.g. in the LAP, a 1-1 mapping between agents and tasks is required - but using SC may lead to two agents selecting the same task).

The main benefit to using SC is that, with a single query to the DNN, a feasible assignment matrix can be recovered (as long as the previous assumptions hold regarding $\mathcal{C}$). In addition, when combined with SC, the DNN can be trained by both supervised learning (SL) and RL.

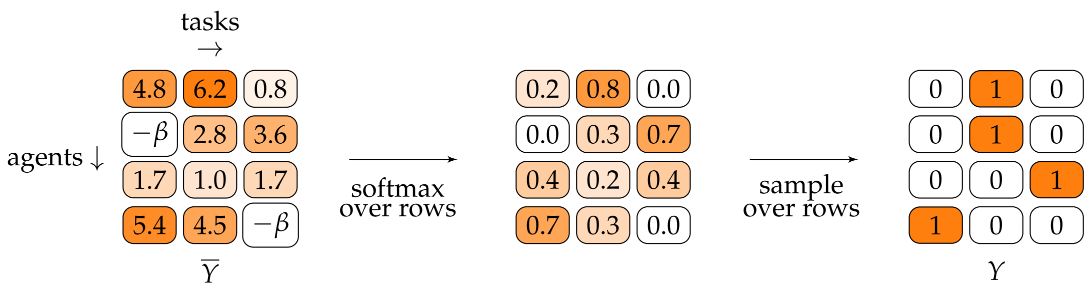
Three significant drawbacks were observed to using SC. As previously noted, for many BAPs, SC is unable to guarantee that the resulting assignment matrix will be feasible as according to $\mathcal{C}$. Second, it was observed that a large number of stacks (>10) are typically required to obtain satisfactory performance on medium size problem instances. Finally, when training with policy-based RL, it can be difficult to effectively coordinate the DNN's actions (as a result of sampling independent PMFs), and so training can be slow and unstable.[2]

[1] Similar operations can also be used if each task is to be selected by a single agent, and an arbitrary number of tasks can be assigned to a single agent.

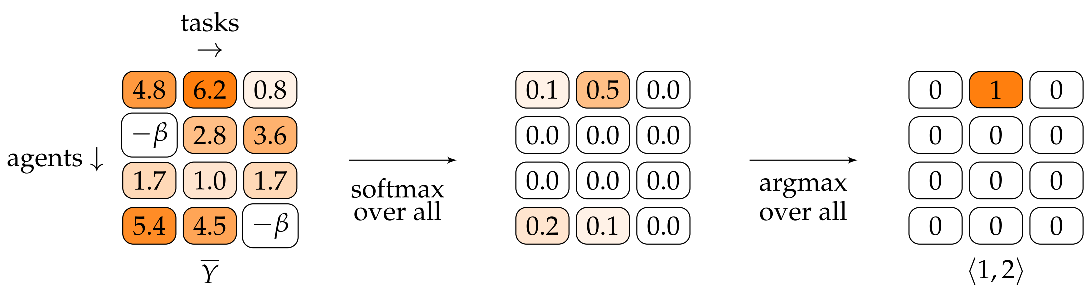
[2] Other RL algorithms, such as Q-Learning, may not suffer from this issue as they do not require an explicitly stochastic policy. However, to keep this thesis concise, such algorithms are not considered.



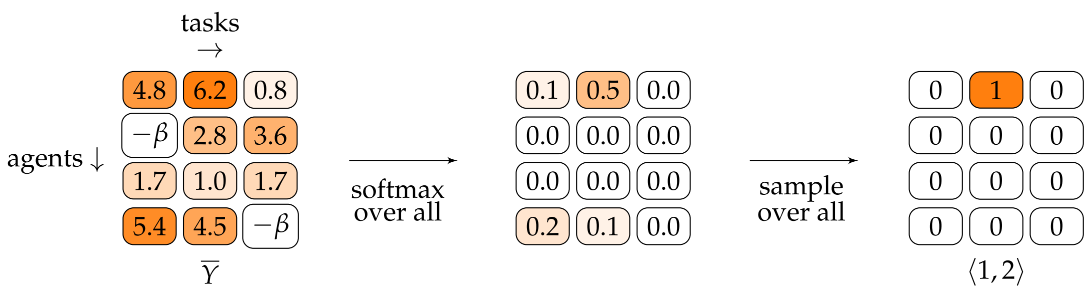
(a) Determinitic simultaneous construction.



(b) Stochastic simultaneous construction.



(c) Deterministic greedy construction.



(d) Stochastic greedy construction.

FIGURE 5.4: Methods of inference for DBA.

Greedy Construction

Rather than simultaneously assigning all agents to tasks in a single round, greedy construction (GC) incrementally builds Y. To use GC, the DNN input X needs to be extended to include an additional binary indicator variable $\mathbb{1}_{i,j}^{\text{assigned}}$, which is equal to unity if agent i has already been assigned to task j. From this newly defined X, DBA is applied in the same way as previously described. It is generally assumed that if $\mathbb{1}_{i,j}^{\text{assigned}} = 1$, then $\mathbb{1}_{i,j}^{\text{infeasible}} = 1$ as that particular agent-task combination has already been selected.

From the output, $\overline{Y}$ is flattened out to yield $\dot{Y} \in \mathbb{R}^{NM}$. This new vector, $\dot{Y}$ has a scalar for each agent-task combination. A softmax operation is applied over the entire vector to yield a PMF over every possible agent-task pair. An agent-task pair can then be selected according to this PMF either deterministically (by taking the argmax) or stochastically (by sampling). From the previous subtraction operation, infeasible agent-task pairs will have zero probability of being selected due to the exponential nature of the softmax operation. After each agent-task selection, it is then necessary to update the new binary indicators variables $\mathbb{1}_{i,j}^{\text{assigned}}$ and $\mathbb{1}_{i,j}^{\text{infeasible}}$ to reflect the new state of the system.

GC has a number of advantages over SC. First, it can be readily applied to a large number of practical BAPs with sophisticated constraints on Y. Second, during preliminary experiments, it was discovered that RL is much more effective when combined with GC rather than SC. It is likely this is because the action space when using GC is dramatically smaller than when using SC (MN vs. M^N). Finally, it was noticed that, when using GC, the resulting DNN requires far fewer stacks than SC. This is likely because the DNN only has to make a single agent-task selection, instead of constructing a complete assignment matrix in a single call.

GC also comes with some disadvantages. First, it is not obvious how GC can be combined with SL. Second, if an optimal assignment matrix has some W number of values equal to unity (that is, $\sum_i \sum_j Y_{i,j}^* = W$), then a DNN with GC requires W separate calls to build the optimal assignment matrix, which potentially makes GC much less efficient than SC.

5.3 Communication Layers

DBA employs communication layers to exchange information across a set of vectors arranged as the rows of a matrix. These communication layers have been specifically designed to address the requirements presented in Chapter 3. This thesis uses the following definition of a communication layer.

Definition 7. *A communication layer C is a function that maps from one matrix to another such that the following properties are satisfied:*

1. *Row maintenance. The number of rows at the output is equal to the number of rows at the input.*

2. *Row dependence. There exists a $Z \in \mathbb{R}^{\Psi \times \Omega}$such that*

$$C(Z) \neq \begin{bmatrix} C(Z_1) \\ C(Z_2) \\ \vdots \\ C(Z_\Psi) \end{bmatrix}, \tag{5.4}$$

where $Z_\ell \in \mathbb{R}^{1 \times \Omega}$ is the ℓ^{th} row of Z.

3. *Row equivariance. If $\rho(Z)$ is a permutation of the rows of Z, then*

$$C(\rho(Z)) = \rho(C(Z)) \tag{5.5}$$

for all possible ρ and Z.

4. *Differentiability. $C(Z)$ is differentiable with respect to input Z. That is, $\nabla_Z C(Z)$ exists.*

Row maintenance is used to assert that the number of objects under consideration (e.g. agents or tasks) is invariant. The row dependence property asserts that computation is performed across the entire matrix Z and that the rows of Z are not treated independently. Row equivariance requires that an equivalent result should be returned regardless of the row permutation of Z. Finally, differentiability is required to allow gradients to flow backwards through the DNN during backpropagation.

The choice of communication layer is a matter of user preference. This thesis presents two communication layers that have shown promising results: pooling and attention.

5.3.1 Pooling

Pooling computes a scalar statistic ζ columnwise across the rows of a matrix input. With input $Z \in \mathbb{R}^{\Psi \times \Omega}$, the resulting row vector $\boldsymbol{\zeta} \in \mathbb{R}^{1 \times \Omega}$ is

$$\boldsymbol{\zeta} = \begin{bmatrix} \zeta((Z^\top)_1) & \zeta((Z^\top)_2) & \dots & \zeta((Z^\top)_\Omega) \end{bmatrix}. \tag{5.6}$$

$\boldsymbol{\zeta}$ is then duplicated and concatenated with Z to yield $C_{\text{pooling}}(Z) \in \mathbb{R}^{\Psi \times 2\Omega}$:

$$C_{\text{pooling}}(Z) = \begin{bmatrix} Z_1 & \boldsymbol{\zeta} \\ Z_2 & \boldsymbol{\zeta} \\ \vdots & \vdots \\ Z_\Psi & \boldsymbol{\zeta} \end{bmatrix}. \tag{5.7}$$

The pooling communication layer is strongly inspired by Sukhbaatar et al., 2016, in which the mean is used to enable communication among cooperative agents. In keeping with Sukhbaatar et al., 2016 (as well as Zaheer et al., 2018 and others), this thesis uses the mean as the pooling scalar statistic.

5.3.2 Attention

Given a set of vectors, an attention mechanism can be applied to extract relevant information. Attention mechanisms are popular in deep learning and are especially prevalent in domains such as natural language processing (NLP) (Vaswani et al., 2017). In the context of NLP, each word in a sentence may be represented by a vector. In order to perform machine translation, attention can be used to query which words are relevant for the purpose of providing the next word in a translated sentence. The attention communication layer is inspired by Deudon et al., 2018, in which attention is used to automatically generate heuristics for the travelling salesman problem.

Canonically, attention uses an external decoder to extract information from a set of vectors. This thesis specifically uses a variant called self-attention. In self-attention, each vector contains its own decoder to make queries about the other vectors undergoing computation.

Given input $Z \in \mathbb{R}^{\Psi \times \Omega}$, self-attention first applies three feedforward operations in parallel to produce queries $Q \in \mathbb{R}^{\Psi \times |K|}$, keys $K \in \mathbb{R}^{\Psi \times |K|}$ and values $V \in \mathbb{R}^{\Psi \times |V|}$. The soft-attention mechanism as described by Vaswani et al., 2017 is then applied:

$$\text{attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{|K|}}\right) V\,. \tag{5.8}$$

The scaling factor $\frac{1}{\sqrt{|K|}}$ is used to counteract the vanishing gradient problem. For further discussion, see Vaswani et al., 2017.

As in Vaswani et al., 2017, this thesis uses multi-head self-attention to perform many attention queries in parallel. Rather than creating a single Q, K and V from Z, h heads are used: $Q = (Q_1, Q_2, \ldots, Q_h)$, $K = (K_1, K_2, \ldots, K_h)$ and $V = (V_1, V_2, \ldots, V_h)$. The self-attention heads are then concatenated together:

$$C_{\text{attention}}(Z) = \begin{bmatrix} H_1 & H_2 & \ldots & H_h \end{bmatrix}\,, \tag{5.9}$$

where

$$H_\ell = \text{attention}(Q_\ell, K_\ell, V_\ell) \tag{5.10}$$

and so $H_\ell \in \mathbb{R}^{\Psi \times |V|}$ and so $C_{\text{attention}}(Z) \in \mathbb{R}^{\Psi \times h|V|}$.

5.4 Learning Algorithms

This thesis considers two families of learning algorithms: supervised learning (SL) and reinforcement learning (RL). SL assumes access to some potentially expensive $f(\mathcal{X})$ that returns the optimal assignment matrix Y^* for any $\mathcal{X}$. SL then trains the DNN to approximate $f(\mathcal{X})$. RL (specifically policy-based RL), evaluates a particular realisation of θ and then uses a gradient approximation in the direction of negative expected objective to update θ . In general, SL is easier to implement and is more likely to find Y^*. RL, however, is more general and can be used to solve problems without requiring a pre-existing method for computing Y^*. SL is more appropriate for accelerating the online execution of a pre-existing method, where RL can be used when no such method exists.

DBA can be trained by both SL and RL. This section builds upon the work presented in Chapter 2 and describes how these learning algorithms can be applied to train the parameters of DBA .

5.4.1 Supervised Learning

It is always assumed that, when using SL, SC is used as the method of inference (as it is not obvious how to combine SL with GC).

Recall that, in SL, there is some potentially expensive $f(\mathcal{X})$ that returns the optimal assignment matrix Y^* for any $\mathcal{X} \in \mathcal{I}$. From the agent PMFs, let $\mathbb{P}_{i,j} \in (0,1)$ be the probability that agent i selects task j and let $Y^*_{i,j} \in \{0,1\}$ indicate whether agent i selects task j according to the optimal assignment matrix Y^*. A cross-entropy loss L_{SL} can then be used to measure the "difference" between the agents' PMFs and the optimal assignment matrix Y^*

$$L_{\text{SL}} = \mathbb{E}\left[-\sum_{i=1}^{N}\sum_{j=1}^{M} Y^*_{i,j}\log(\mathbb{P}_{i,j})\right]. \tag{5.11}$$

The above expectation is taken uniformly over all problem instances $\mathcal{X} \in \mathcal{I}$. In practice, the gradient is estimated $\widehat{L_{\text{SL}}} \approx L_{\text{SL}}$ using samples and stochastic gradient descent is used to iteratively improve θ.

5.4.2 Reinforcement Learning

RL can be used to train DBA, regardless of whether SC or GC is used. As previously noted, RL is usually more performant when combined with GC, but in principle, either method of inference can be used.

Simultaneous Construction

After the softmax operation, a PMF over all possible assignment matrices in $\mathcal{Y}$ can be derived. If $\mathbb{P}_{i,j} \in (0,1)$ is the probability that agent i selects task j, then the probability that a particular assignment matrix Y (where agent i selects task j_i) is stochastically constructed is given by

$$\begin{aligned}\mathbb{P}(Y) &= \mathbb{P}(Y_1 = j_1, Y_2 = j_2, \ldots, Y_N = j_N)\\ &= \prod_{i=1}^{N}\mathbb{P}(Y_i = j_i)\\ &= \prod_{i=1}^{N}\mathbb{P}_{i,j_i}.\end{aligned} \tag{5.12}$$

And so, a complete assignment matrix can be sampled as a single action u, where the set of all actions is simply $\mathcal{U} = \mathcal{Y}$. As SC yields a complete assignment matrix, an objective can be calculated as according to $J(\mathcal{X}, Y)$. The following information is now recorded: an action

Y, a differentiable probability of selecting action Y (given by $\mathbb{P}(Y)$), and a real-valued objective $J(\mathcal{X}, Y)$. By collecting many data points, the policy gradient can be approximated. Recall that, from 2.2.1

$$\nabla_\theta J(\pi_\theta) \approx \widehat{\nabla_\theta J(\pi_\theta)} = \frac{1}{|\mathcal{T}|} \sum_{\mathcal{T}} g \nabla \log \pi_\theta(u|s), \tag{5.13}$$

where $\nabla \log \pi_\theta(u|s)$ is the gradient of the probability of selecting a particular action from a particular state and g is the return after taking said action. Substituting in the values from SC yields

$$\nabla_\theta J(\pi_\theta) \approx \widehat{\nabla_\theta J(\pi_\theta)} = \frac{1}{|\mathcal{T}|} \sum_{\mathcal{T}} J(Y, \mathcal{X}) \nabla \log \mathbb{P}(Y). \tag{5.14}$$

With the above equation, stochastic gradient ascent can be used (or descent if minimisation of $J(\mathcal{X}, Y)$ is required) to iteratively move the DNN parameters in the direction of the expectation of the objective function.

The above expression only uses the objective (or the return) to estimate the policy gradient. Such an estimation is of high variance (recall 2.2.2). However, a parameterised critic $v_\phi^{\pi_\theta}(X)$ can be used to estimate the average objective that the policy will receive in expectation. The critic has the same architecture as the policy but uses its own parameters (ϕ as opposed to θ). At the output, the critic simply averages over all of the entries of $\overline{Y}_\phi$ to yield $v_\phi^{\pi_\theta}(X)$. If the value from the parameterised critic is accurate, that is, it gives a good approximation of the expected objective from the current policy given the problem instance $\mathcal{X}$, then the critic can be used as a baseline to construct a lower-variance estimate of the policy gradient

$$\nabla_\theta J(\pi_\theta) \approx \widehat{\nabla_\theta J(\pi_\theta)} = \frac{1}{|\mathcal{T}|} \sum_{\mathcal{T}} \left(J(Y, \mathcal{X}) - v_\phi^{\pi_\theta}(X) \right) \nabla \log \mathbb{P}(Y). \tag{5.15}$$

Initially the critic will give uniformed values that do not help make the policy gradient estimate more accurate. However, SL is used to train the critic to approximate the value function. For details, see 2.2.4.

Greedy Construction

The sequential behaviour of GC naturally gives rise to a sparse-reward Markov Decision Process (MDP). Therefore, given a reward function, any RL algorithm can be applied to gradually improve the DNN's parameters. This thesis uses advantage actor-critic (A2C) paired with generalised advantage estimation (GAE), as described in 2.2. Note however, that there exist many different RL algorithms that could be used instead.[3]

This thesis uses a critic $v_\phi^{\pi_\theta}$ similar to that described previously. Remember that, X needs to be expanded to include the additional binary indicator variable $\mathbb{1}_{i,j}^{\text{assigned}}$. Again, the value

[3]So long as they can handle discrete action spaces - which is not possible with specifically continuous algorithms such as Deep Deterministic Policy Gradients (Silver et al., 2014).

function is recovered by averaging over all values of $\overline{Y}_\phi$,

$$v_\phi^{\pi_\theta}(X) = \frac{1}{NM} \sum_i \sum_j \overline{Y}_{\phi,i,j} \tag{5.16}$$

To completely implement an algorithm such as A2C, a reward function needs specifying. For all of the applications in this thesis, the following sparse reward function is used:

$$r(\mathcal{X}, Y_{i,j}) = \begin{cases} 0 & \text{if } \sum_i \sum_j \mathbb{1}_{i,j} < NM \\ J(\mathcal{X}, Y) & \text{if } \sum_i \sum_j \mathbb{1}_{i,j} = NM. \end{cases} \tag{5.17}$$

Summary

In this chapter, a novel deep learning architecture entitled Deep Bipartite Assignments (DBA) was presented. DBA is specifically designed for representing and optimising BAPs. Through the use of communication layers, DBA satisfies the specifications set out in Chapter 3, achieving both permutation equivariance and parameter independence to variable N and M. In the upcoming chapters, DBA is validated over two NP-Hard BAPs: the weapon-target assignment (WTA) problem and the multi-resource generalised assignment problem (MRGAP).

Chapter 6

Application 1: The Weapon-Target Assignment Problem

As an initial application, DBA was tested on the weapon-target assignment (WTA) problem. The WTA problem was selected because it is non-trivial but does not impose particularly sophisticated constraints on the set of feasible assignment matrices. The relatively simple constraints of the WTA problem allow us to first verify that Deep Bipartite Assignments (DBA) is effective through the use of supervised learning (SL) in combination with simultaneous construction (SC). Recall that, from the previous chapter, SL is generally considered to be much more stable than reinforcement learning (RL). Therefore, with SL, it is possible to quickly validate that DBA is capable of finding optimal assignment matrices given an arbitrary problem description.

The chapter begins by defining the WTA problem and presenting relevant background literature. Two baseline algorithms are presented to assess the relative performance of DBA . Finally, DBA is demonstrated to be competitive with these baselines, regardless of whether it is trained by SL or RL.

6.1 Background

The WTA problem was first formally stated by Manne, 1958 and is loosely modelled on a military engagement. WTA assume a battle scenario with a fleet of weapons (or agents) and a set of targets (or tasks). The objective is to minimise the expected surviving values of the targets. Each weapon-target combination has a kill probability

$$p_{i,j} = \mathbb{P}(\text{target } j \text{ is destroyed by weapon } i \mid Y_{i,j} = 1) \tag{6.1}$$

and each target has positive value $v_j > 0$.

The objective function J_{WTA} assumes that the kill-probabilities $p_{i,j}$ are independent of one another. Therefore, the probability that a given target survives is given by the product of conjugate probabilities $\prod_i (1 - p_{i,j})^{Y_{i,j}}$. The solution to a WTA problem instance is an optimal assignment matrix Y^* such that

$$Y^* = \min_Y J_{\text{WTA}} = \min_Y \sum_{j=1}^{M} v_j \prod_{i=1}^{N} (1 - p_{i,j})^{Y_{i,j}} . \tag{6.2}$$

For the WTA constraints $\mathcal{C}_{\text{WTA}}$, it is assumed that each weapon can be assigned to a single target (but a single target can be selected by an arbitrary number of weapons). This leads to the following set of constraints on Y for all i,

$$\sum_j Y_{i,j} = 1\,. \tag{6.3}$$

The WTA problem is a bipartite assignment problem (BAP) as according to Definition 2 as its problem instances can be represented by the tuple $\mathcal{X} = \langle A, T, P, E \rangle$, where,

- A is unused $(|A| = 0)$.
- T is reduced to an M entry vector, where each element represents the value of the j^{th} target. That is, for task j, $T_j = v_j$ and so $|T| = 1$.
- P is reduced to an $N \times M$ matrix, where each element $P_{i,j} \in [0, 1]$ represents the kill-probability that agent i will destroy target j if it is assigned to it. For each agent-task pair, $P_{i,j} = p_{i,j}$ and so $|P| = 1$.
- E is unused $(|E| = 0)$.

The WTA problem is well-known to be NP-complete and so cannot be solved exactly in polynomial time (Lloyd and Witsenhausen, 1986) . Lower bounding strategies can be used to find exact solutions for medium-sized instances (e.g. 20 agents and 20 tasks), but no exact algorithms exist for finding real-time solutions to large problem instances (e.g. 100 agents and 100 tasks). There are well-known heuristics for finding near-optimal solutions to the WTA problem (Ahuja et al., 2003). These heuristics are known to generate fast solutions within a few percentage points of optimality. Therefore, despite being NP-complete, the WTA problem is essentially solved for practical purposes. DBA is not necessarily designed to compete with existing heuristics for the WTA problem. The WTA problem is instead a benchmark to illustrate how DBA can be applied to a practical, non-trivial example.

6.2 Design Modifications

The relatively modest constraints on Y means that either simultaneous construction (SC) or greedy construction (GC) can be used as a method of inference for DBA . To verify that DBA is capable of constructing optimal assignment matrices, SC was combined with SL (SC-SL). Later, in other experiments, DBA was trained using GC with RL (GC-RL) to test whether DBA can discover near-optimal assignment matrices from scratch without human supervision. In both cases, independent experiments were carried out with both of communication layers presented in 5.3 (pooling and attention).

6.3 Baselines

Two baselines are supplied for the WTA problem: a branch and bound algorithm and a genetic algorithm (GA). The branch and bound comes from Ahuja et al., 2003, and was used to generate a dataset of exact, optimal solutions to the WTA problem. These exact solutions can be used to verify the relative optimality of DBA . The branch and bound however, requires expert human knowledge about the structure of the WTA problem to derive and implement. GAs, on the other hand, can be readily applied to a large number of different problems with little human supervision. In this way, a GAs are a suitable comparison with DBA as they are general-purpose and are not limited to a particular problem definition. Ideally, DBA should be competitive with the branch and bound, while outperforming the GA.

6.3.1 Branch and Bound

The branch and bound uses a depth-first search to iteratively construct an assignment matrix, one agent-task pair at a time. The algorithm begins with an empty assignment matrix. At each step of the algorithm, an agent-task pair is added to the current assignment matrix. If the current assignment matrix is complete (that is, no more agent-task assignments can be added without violating the WTA constraints), then the objective is computed according to J_{WTA}. Throughout the branch and bound, the best objective and the best corresponding assignment matrix are tracked.

Backtracking is used to search through different assignment matrices. Backtracking just means that, the most recently added agent-task pair is removed and a new agent-task pair is added in its place. If all new agent-task pairs have been exhausted, then the second-to-last added agent-task pair is removed and a new agent-task pair is added in its place. If this process repeats until there is an empty assignment matrix, and no new agent-task pairs can be added that have not used been previously, the branch and bound is terminated.

As each agent-task pair is added, a lower bound is computed on the current assignment matrix. This lower bound describes the best possible objective that can be obtained by adding agent-task pairs to the current assignment matrix. Backtracking is induced by two events: whenever a complete assignment matrix is found, or whenever the current lower bound is larger than the best objective on record. If a computed lower bound is larger than a previously found objective, then there is no need to keep adding agent-task pairs to the current assignment matrix (as it is impossible to beat the best assignment matrix on record). Without lower bound computation, a branch and bound degenerates into depth-first exhaustive search.

Ahuja et al., 2003 state three possible lower bounds for the WTA problem. This chapter uses their maximum marginal return-based (MMR) lower bounding scheme as it is the easiest to understand and implement. The MMR scheme assumes that, rather than having a unique kill probability for every agent-task pair, a vector of kill probabilities is constructed using the best kill-probability for each task. The current (partial) assignment matrix can then be completed by selecting agent-task pairs to improve upon the current objective in a greedy manner using the new vector of kill probabilities. This scheme always gives a lower bound on the best possible objective using the current assignment matrix. For a proof of this result, see Ahuja et al., 2003.

Ahuja et al., 2003 state that, with MMR lower-bounding, exact solutions can be found when $N = M = 20$ in less than a second. However, the authors only performed their tests on individual problem instances. This statistic may be misleading as solution speed appears to strongly depend upon the structure of the particular problem instance. In the experiments carried out for this thesis, solving instances with $N = M = 20$ took between one second and five minutes (depending on the instance). However, the code is mostly written in pure Python (which is notoriously slow) and the implementation likely contains inefficiencies. To speed up the branch and bound, an initial solution was constructed using a heuristic from the same paper. The heuristic, called the "minimum cost flow construction heuristic", in essence, uses the solution to a closely related network flow problem to quickly find a good initial solution (which, in fact, is usually optimal when $N = M = 20$).

6.3.2 Genetic Algorithm

A GA is a simple, derivative-free technique that combines random search with biologically-inspired heuristics. For an introduction to genetic algorithms, see Eiben and Smith, 2015. GAs provide a suitable comparison to DBA because they are also black-boxes that do not require any human knowledge to find heuristic solutions. Unlike many other optimization paradigms, GAs can be easily applied in non-convex settings with integer variables, which make them ideal for many assignment problems. GAs are also extremely parallelisable, and so can easily take advantage of large, multi-core systems. A key difference between DBA and the GA is that, once trained, DBA can instantaneously finds near-optimal solutions for new problem instances, where the GA must essentially start from scratch for every new problem instance. GAs are well known to have two other issues: 1) they require many evaluations of the objective function (and so can only be used in applications where the objective function can be evaluated cheaply) and 2), they tend to scale poorly as the size of the optimisation variable increases (which, here, is the number of rows in the assignment matrix Y).

A high-quality GA implementation from the Pygmo library (Biscani et al., 2010) was used as the baseline. The Pygmo library automatically parallelises the GA to run on all available cores (12 in the experiments for this chapter) and has an efficient backend written in C++.

The Pygmo GA uses the following procedure. The GA begins with a large population of random assignment matrices. Let the size of the population be given by $\overline{ps}$. Each assignment matrix is captured as a vector of length N, with integer values between 1 and M. The WTA objective is then computed for all assignment matrices in parallel. The Pygmo GA then employs three operations: selection, crossover and mutation. Each operation is performed $\overline{ps}$ in parallel to produce a new population of $\overline{ps}$ assignment matrices. At selection, μ_s random assignment matrices are sampled, and the one with the best objective is selected. At crossover, two random assignment matrices are selected (in the form of agent-wise vectors) denoted as parent and the partner respectively. A random point in the parent vector is selected. From this random point until the end of the vector, each element is replaced with the corresponding element from the partner vector with probability μ_c. Finally, mutation randomly perturbs each element of the resulting vector with probability μ_m. There are now $2\overline{ps}$ assignment matrices. The GA then uses a technique called reinsertion to isolate the best $\overline{ps}$ assignment matrices, which will be carried forward into future iterations of the algorithm. The algorithm iterates continually until some pre-defined time limit is reached.

A hyperparameter search was conducted to optimise μ_s, μ_c, and μ_m but were unable to outperform the default hyperparameters given by the Pygmo implementation. For all of the experiments, $\overline{ps} = 512$.

6.4 Experiments

DBA was applied to non-trivial instances of the WTA problem with $N = 20$ agents and $M = 20$ tasks. The experiments followed the modelling assumptions of Ahuja et al., 2003 by sampling $p_{i,j}$ uniformly from $[0.6, 0.9]$ and v_j as uniform random integers from $[25, 100]$.

DBA was trained with SC-SL and GC-RL. For both approaches, experiments were performed with both of the communication layers presented in Section 5.3. Thus, four variants were trained in total: pooling with SL (P-SL), attention with SL (A-SL), pooling with RL (P-RL) and attention with RL (A-RL). Table 6.1 details some of the experimental settings.

TABLE 6.1: Settings for each variant.

Setting	P-SL	A-SL	P-RL	A-RL
Number of stacks S	18	12	3	2
Number of parameters	654529	103873	191362	9218
$\lvert\mathcal{E}\rvert$	64	32	64	16
$h, \lvert K\rvert, \lvert V\rvert$	n/a	4, 8, 8	n/a	4, 4, 4
Initial learning rate α	10^{-4}	10^{-3}	5×10^{-4}	10^{-4}

For the SL experiments, a training dataset with 200,000 optimal examples consisting of $\langle \mathcal{X}, Y^* \rangle$ pairs was created. For both learning algorithms, an additional 1,000 optimal examples were created for evaluation and a final 1,000 optimal examples for testing. The evaluation dataset was used to determine the best parameters θ^* from each training run. The test dataset was then used to make conclusions about how well each trained DBA generalised to problem instances that were not seen during training or evaluation. Optimal assignment matrices Y^* were generated using the minimum cost flow construction heuristic from Ahuja et al., 2003 and optimality was asserted by the branch and bound from 6.3.1.

It was observed that a large number of optimal examples were required to ensure generalisation when training with SL. 200,000 examples led to generalisation, where 50,000 examples led to overfitting (where DBA learnt to memorise the training dataset but performed poorly on the evaluation and test datasets).

Throughout the experiments, many of the current best practices from deep learning were adopted. The inputs to DBA were normalised using the previously stated modelling assumptions. Batch normalisation was added after every major operation (Ioffe and Szegedy, 2015). The Adam update rule (Kingma and Ba, 2015) with an exponentially decaying learning rate was used instead of vanilla stochastic gradient descent. Skip connections were included for the pooling-based architectures (Sukhbaatar et al., 2016) and residual connections were used for the attention-based architectures (He et al., 2016).

For all of the nonlinearities, the rectified linear unit $\sigma(\cdot) = \text{relu}(\cdot)$ was used. For P-RL, 10 problem instances were batched together for each parameter update. For the other variants, 20 problem instances were batched together for each parameter update. For GAE, $\gamma = 1.0$ and $\lambda = 0.99$ (see 2.2.3 Schulman et al., 2016). All hyperparameters were found by random search.

All experiments were run in Python and TensorFlow on an Ubuntu 16.04 machine with six Intel(R) Core(TM) i7-8700K CPUs @ 3.70GHz and an Nvidia GTX 1080 GPU.

The primary metric of interest was the optimality gap $\overline{og}$, defined as:

$$\overline{og} = 100 \times \frac{J(Y, \mathcal{X}) - J(Y^*, \mathcal{X})}{J(Y^*, \mathcal{X})}\%\,. \tag{6.4}$$

During each training run, the mean optimality gap was tracked over the evaluation dataset. DBA was evaluated on the evaluation dataset every 100 parameters updates when training with SL and every 50 updates when training with RL. To ensure reproducibility, 50 training runs were performed for each communication layer/learning algorithm combination. For each training run, the DBA parameters θ^* that minimised the mean optimality gap over the evaluation dataset were saved. Table 6.2 displays the average number of parameter updates required to find θ^* for each variant.

TABLE 6.2: Training time for each variant averaged over 50 independent runs.

Metric	P-SL	A-SL	P-RL	A-RL
Number of updates to find θ^*	46839	41643	2024	4450
Time per update (seconds)	0.060	0.067	0.245	0.423

With the best parameters from each training run, the following metrics were computed over the test dataset:

- Mean gap: The average optimality gap over the entire test dataset.
- % optimal: How often Y^* was found as a percentage of the entire test dataset.
- % upper bound (u.b.): The value for which a percentage of assignments had a lower optimality gap.

Numerical results are reported in Table 6.3 and empirical cumulative distribution functions (CDFs) are plotted in Fig. 6.1.

The attention-based architectures yielded the best results, with significantly fewer parameters than their pooling-based counterparts. The SL-based architectures were able to find optimal assignment matrices for the vast majority of problem instances in the test dataset. In contrast, the RL-based architectures were rarely able to find the optimal assignment matrices, but tended to have more reliable performance across the entire test dataset (see Fig. 6.1).

A-RL is perhaps the most impressive variant, as it was able reliably generate near-optimal solutions for the entire test dataset and did not require any optimal demonstrations to train.

6.4.1 GA Comparison

The GA was applied to the test dataset. The GA was given access to all available computational resources and its performance was benchmarked for various time limits (up to 10 seconds). It

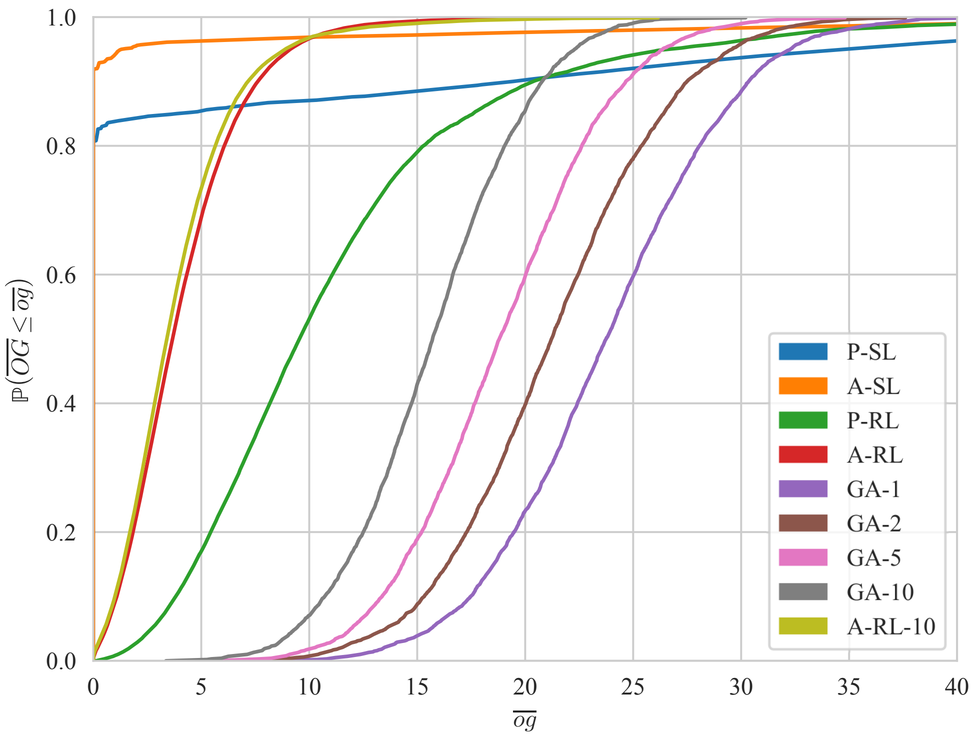


FIGURE 6.1: Empirical CDFs for all of the methods from this work. $\overline{OG}$ is a random variable that represents the optimality gap for a random problem instance from the test dataset.

was found that the best performance was achieved with a population size of 512, with the rest of the settings unchanged from the default values recommended by Pygmo. To ensure fairness, the GAs metrics were averaged over four independent test runs. The results are reported in Table 6.4 and empirical CDFs are plotted in Fig. 6.1.

The GA was unable to outperform A-RL on any of the observed metrics and had a worse mean optimality gap than any of the DBA variants. Significant compute was required for the GA to even be competitive. For example, with a 10 second time limit, the GA (GA-10) performed approximately 3×10^6 objective function evaluations for each individual problem instance. A-RL however, produced better quality results in less than 50ms without any online objective function evaluation. In fact, throughout the entire training process, A-RL performed fewer objective function evaluations than GA-10 performed for each individual problem instance. For example, the best A-RL parameters were found after less than 1.8×10^5 objective function evaluations on average (of which half of the function evaluations were used to track the mean optimality gap over the evaluation dataset and were not directly involved in parameter updates).

6.4.2 Scalability

The experiments so far have been limited to WTA problem instances with $N = M = 20$. To verify that DBA is not severely limited by increasing N and M, an initialised DBA model was tested with large WTA problem instances. Table 6.5 displays query-times for various values of $N = M$. Each time is averaged over 100 random queries. Even in the worst case with A-RL and $N = M = 128$, an assignment matrix is returned in just over two seconds. Although the RL-based architectures are shallower than the SL-based architectures (see Table 6.1), the query-times are significantly worse. This is because N calls need to be made to the DNN to cycle through the complete MDP. Note that if very large values of N are required, SC-RL can be used to compute a complete assignment matrix with a single call to the DNN. However, in the preliminary experiments, it was noticed that performance suffered when using such an approach. For example, with A-RL, a mean optimality gap of ≈12% was observed when using SC as compared to ≈4% when using GC.

A key feature of DBA is that the parameters θ are not explicitly conditioned on N or M. Therefore, a pre-trained DBA can be applied to problem instances with variable N and M without alteration. To demonstrate this feature, A-RL was trained on a randomised WTA training dataset with $N = M = 10$. This approach is referred to as A-RL-10. A-RL-10 was then applied to the original test dataset with $N = M = 20$ without any additional training. The test results are reported in Table 6.6 and the CDF is plotted in Fig. 6.1.

TABLE 6.3: Results on the test dataset for each variant averaged over 50 training runs.

Metric	P-SL	A-SL	P-RL	A-RL
Mean	4.562	**1.128**	11.247	4.124
% optimal	84.007	**93.900**	0.195	1.954
50% u.b	**0.000**	**0.000**	9.549	3.659
90% u.b.	19.438	**0.001**	20.365	7.615
95% u.b.	34.672	**1.304**	26.672	9.119
99% u.b.	61.184	40.338	41.261	**13.111**
100% u.b.	155.341	80.106	61.932	**24.915**

TABLE 6.4: GA results on test dataset for various time-limits averaged over four runs.

Metric	GA-1	GA-2	GA-5	GA-10
Mean gap	23.809	21.317	18.944	15.749
% optimal	0.000	0.000	0.000	0.000
50% u.b	23.752	21.217	18.801	15.712
90% u.b.	30.340	27.487	24.656	20.782
95% u.b.	32.269	29.576	26.371	22.290
99% u.b.	36.675	33.233	30.040	24.954
100% u.b.	41.083	37.608	34.808	30.209

TABLE 6.5: Average query-time (in seconds) for WTA instances with variable N, M.

$N = M$	P-SL	A-SL	P-RL	A-RL
2	0.005	0.013	0.003	0.005
4	0.006	0.011	0.005	0.010
8	0.006	0.012	0.011	0.018
16	0.006	0.012	0.023	0.039
32	0.008	0.012	0.053	0.078
64	0.018	0.022	0.255	0.290
128	0.059	0.080	1.572	2.151

TABLE 6.6: Results on test dataset for A-RL-10 averaged over 100 training runs.

Metric	A-RL-10
Mean gap	3.909
% optimal	1.940
50% u.b	3.396
90% u.b.	7.158
95% u.b.	8.878
99% u.b.	14.892
100% u.b.	26.158

Surprisingly, A-RL-10 outperformed A-RL across a number of metrics while also being much faster to train. A similar number of parameter updates were required, but each update only took 0.12 seconds, as opposed to 0.42 seconds previously. Therefore, A-RL-10 was able to generate good results on the test set with less than 10 minutes of training. In the future, the pseudo-invariance of DBA with respect to N and M may help us scale DBA to more difficult assignment problems.

Summary

DBA can generate fast, near-optimal solutions for non-trivial instances of the WTA problem. It was demonstrated that, when contrasted against a comparable black box method, DBA returns faster, better quality results with significantly less compute. Finally, it was shown that DBA can generalise to larger assignment problems than those seen during training.

Chapter 7

Application 2: The Multi-Resource Generalised Assignment Problem

In this chapter, DBA is applied to a more difficult problem: the multi-resource generalised assignment problem (MRGAP). The MRGAP is NP-hard and is APX-hard just to approximate (Martello and Toth, 1990). Here, with an almost identical approach to the previous chapter, Deep Bipartite Assignments (DBA) is competitive with an effective, well-known baseline.

7.1 Background

Th MRGAP is a generalisation of the classic knapsack problem (KP) from computer science (Martello and Toth, 1990). Consider a single agent with many available tasks. Each task has a profit p_j and a weight w_j. In addition, the agent has a capacity c. The agent can select any number of tasks, so long as the sum of the weights of the selected tasks does not exceed its capacity. Formally, the KP seeks to maximise

$$J_{\mathrm{KP}} = \sum_j p_j Y_j \,, \tag{7.1}$$

subject to

$$\sum_j w_j Y_j < c \,. \tag{7.2}$$

If, somehow, the agent can select fractional tasks (that is $Y_j \in [0,1]$), then the KP can be solved with a trivial greedy algorithm. However, the more interesting case in which Y_j is constrained to binary values is NP-Hard (Martello and Toth, 1990).

The KP can be extended to the generalised assignment problem (GAP). Instead of considering a single agent, the GAP has N agents. Each agent has its own capacity c_i and each agent-task pairing has a profit $p_{i,j}$ and a weight $w_{i,j}$. The GAP requires the maximisation of

$$J_{\mathrm{GAP}} = \sum_i \sum_j p_{i,j} Y_{ij} \,, \tag{7.3}$$

subject to

$$\sum_j w_{i,j} Y_{i,j} \le c_i \tag{7.4}$$

for all i and

$$\sum_i Y_{i,j} \leq 1 \tag{7.5}$$

for all j.

Another variant on the KP is the multidimensional knapsack problem (MKP). In the MKP, each task has a K-dimensional vector of weights and the agent has a K-dimensional vector of capacities. That is, $w_j, c \in \mathbb{R}^K$. The objective function $J_{\text{MKP}} \equiv J_{\text{KP}}$. However, the constraints become

$$\sum_j w_{j,k} Y_j < c_k \tag{7.6}$$

for all $k \in \{1, 2, \ldots, K\}$.

Finally, the MRGAP generalises the MKP in the same way that GAP generalises KP to multiple agents. Given N agents, maximise

$$J_{\text{MRGAP}} \equiv J_{\text{GAP}} = \sum_i \sum_j p_{i,j} Y_{ij}\,, \tag{7.7}$$

subject to

$$\sum_j w_{i,j,k} Y_{i,j} \leq c_{i,k} \tag{7.8}$$

for all $k \in \{1, 2, \ldots, K\}$ and all i, and

$$\sum_i Y_{i,j} \leq 1 \tag{7.9}$$

for all j.

The MRGAP is a bipartite assignment problem as according to Definition 2 as its problem instances can be represented by the tuple $\mathcal{X} = \langle A, T, P, E \rangle$, where,

- A is a capacity vector for each agent ($|A| = 0$). That is, for agent i, $A_i = c_i = [c_{i,1}, c_{i,2}, \ldots, c_{i,K}]^\top$ and $|A| = K$.
- T is unused, as each agent-task combination is unique ($|T| = 0$).
- P is a three dimensional array of agent-task information. For each agent-task pair, $P_{i,j} = [p_{i,j}, w_{i,j,1}, w_{i,j,2}, \ldots, w_{i,j,K}]^\top$ and so $|P| = K + 1$.
- E is unused ($|E| = 0$).

From the MRGAP, all of the other problems (KP, GAP, MKP) can be derived as special cases. Like all of the previously mentioned problems, a branch and bound can be used to find exact solutions, but this becomes impractical as the problem instances become larger. During preliminary experimentation, it was noticed that branch and bound scale especially poorly as K increases.

7.2 Baseline

The greedy heuristic from Cohen et al., 2006 was chosen as a baseline for the MRGAP (denoted as GH). The GH was chosen as a baseline because it is simple to implement while maintaining appealing theoretical guarantees.

The GH uses a KP solver as a subroutine. The KP solver can either be exact or a heuristic. The GH then uses the following algorithm. First, arbitrarily select an agent. Then, solve the corresponding KP for that particular agent and update the assignment matrix to reflect the tasks chosen by the agent. Next, another agent is selected and the process is repeated. However, before the KP solver is called again, the profits are scaled to reflect the current assignments. For example, if a given agent-task combination has profit $p_{i,j}$ and the task has been selected by another agent ℓ, then the residual profit $p'_{i,j} = p_{i,j} - p_{\ell,j}$ is computed. If the KP solver still attempts to select task j given this residual profit, then task j is moved from agent ℓ to agent i.

The GH has a running time of $\mathcal{O}(Nf(M) + NM)$, where $\mathcal{O}(f(M)$ is the running time of the KP solver. Interestingly, the GH guarantees that the final assignment matrix will have an objective that is, at least $\frac{1}{1+\alpha}$ of the optimal objective. That is,

$$J_{\text{GAP}}(\mathcal{X}, Y_{GH}) \geq \frac{J_{\text{GAP}}(\mathcal{X}, Y^*)}{1+\alpha} \tag{7.10}$$

for all $\mathcal{X} \in \mathcal{I}_{\text{GAP}}$, where α is the approximation ratio of the KP solver. Therefore, in the case that the KP solver is optimal ($\alpha = 1$), the assignment matrix returned by the GH will have an objective that is no worse than half the optimal objective.

In the original algorithm, the KP solver is invoked N times, once for each agent. However, if an agent i has a task removed from it by another agent ℓ, then it may have enough freed up capacity to accommodate additional tasks. Therefore, the GH can be strengthened by continuing to invoke the KP solver over each agent with updated residual profits until some notion of convergence is reached (for example, if Y stays constant throughout N consecutive rounds of invoking the KP solver). This improved strategy is used as the baseline in this chapter.

Note that the GH was designed for the original GAP (with $K = 1$). However, it is reasonable to use this algorithm in the context of MRGAP as long as an appropriate MKP solver is supplied.

7.3 Design Modifications

DBA can be used with minimal alteration from the previous chapter. The only major difference is that, the set of assignment constraints $\mathcal{C}$ has changed. Unlike the previous chapter, agents can continually select tasks so long as they have enough capacity (rather than being limited to one task). In addition, each task can be selected by at most a single agent. These constraints make using simultaneous construction (SC) difficult, as it is hard to guarantee that the resulting assignment matrix is always feasible. Therefore, in the experiments, the only inference method/learning algorithm is greedy construction (GC) paired with reinforcement learning (RL).

7.4 Experiments

7.4.1 Preliminary Experiments

Upon first exposing DBA to the MRGAP, it was observed that the pooling communication layers performed significantly better their attention-based counterparts. P-RL required far fewer training iterations and produced higher quality assignments than A-RL. This result is somewhat surprising. Recall that, in the previous chapter, A-RL convincingly outperformed P-RL (though note that even then, A-RL generally took longer to converge than P-RL - see Table 6.2). The core experiments therefore focus on just using P-RL.

Early on, it was also noticed that DBA is not pseudo-invariant with respect to the size of MRGAP instances. Recall that, when trained on the WTA problem with $N = M = 10$, A-RL was able to achieve almost identical performance to being trained on $N = M = 20$. Here, DBA can still generalise to larger problem instances than those seen during training, but optimal performance was always achieved when the model was trained over the same distributions of N and M as found in the test dataset.

7.4.2 Core Experiments

DBA was trained over a number of different MRGAP configurations. All hyperparameters were left unchanged from the previous chapter (see Table 6.1). Better results may be possible with further hyperparameter exploration, but the hyperparameter settings were left unchanged to show the robustness of DBA to new problem settings. To gain a thorough understanding of how well DBA scales to different sized problem instances, the following Cartesian product over MRGAP parameters was considered,

$$\{(N, M, K) : N \in \{2, 4, 8\}, M \in \{10, 20, 40\}, K \in \{1, 2, 3\}\} . \tag{7.11}$$

Profits $p_{i,j}$ and weights $w_{i,j,k}$ were sampled uniformly from the integers between 1 and 100. The capacities $c_{i,k}$ were sampled uniformly from the integers between $\frac{100M}{8}$ and $\frac{100M}{4}$.

As a baseline, the GH was as outlined in 7.2 was used. Google's OR Tools MKP solver was used as a subroutine for the GH. This MKP solver uses a branch and bound to obtain exact solutions to MKP problem instances. Therefore, the GH always find assignments with objectives that are, at worst, half the optimal objective (as $\alpha = 1$).

For each (N, M, K) tuple, an evaluation dataset and a test dataset were created with 200 and 1,000 problem instances respectively. DBA was then trained over each (N, M, K) tuple and saved whichever parameters maximised the average objective over the evaluation dataset. The GH was run over each test problem instance to obtain a series of $(\mathcal{X}, Y_{\text{GH}})$ pairs. For each set of trained DBA parameters and every relevant test problem instance, the following objective gap was computed $\overline{og}$,

$$\overline{og} = \frac{J_{\text{MRGAP}}(\mathcal{X}, Y) - J_{\text{MRGAP}}(\mathcal{X}, Y_{\text{GH}})}{J_{\text{MRGAP}}(\mathcal{X}, Y_{\text{GH}})} \times 100\% . \tag{7.12}$$

A positive objective gap $\overline{og} > 0$ indicates that DBA model found a better assignment matrix than the GH, while an objective gap of 0 $\overline{og} = 0$ indicates that the DBA model found an equivalent assignment to the GH.

For each (N, M, K) tuple, the following statistics were recorded over the test dataset:

- The average objective gap.
- The percentage of assignments that had a better objective than the GH (denoted $> GH$ in the results table).
- The percentage of assignments that had an equal objective than the GH (denoted $= GH$ in the results table).
- The percentage of assignments that had a worse objective than the GH (denoted $< GH$ in the results table).

12 DBA models were trained for each (N, M, K), combination. The results are displayed in Table 7.1. For each test problem instance, both the average and the best objective obtained by the 12 DBA models were recorded. Taking the best assignment matrix from 12 separate models is justified in this case as each DBA model is independent and can be queried in parallel with all of the others. In addition, the MRGAP objective can be computed very cheaply, and so does not add significant overhead to the entire process. This approach, of using a number of independent deep neural networks (DNNs) to find a number of different solutions to a combinatorial optimisation problem, is taken from Bello et al., 2016. Such an approach is valid so long as the objective can be computed relatively cheaply, and that there are enough available computational resources to query all trained DNNs in parallel. Note that a similar approach could also be applied to the WTA problem from Chapter 6, resulting in monotonic improvement to the objective function.

DBA is competitive with the GH across all combinations of N, M and K, with objective gaps around 1% in absolute value. In certain scenarios, for example $N = 4, M = 10$, DBA outperforms the GH across the majority of the test dataset. However, in other cases, for example, whenever $N = 8$, DBA tends to perform slightly worse than the GH.

The performance of DBA is a somewhat unpredictable, nonlinear function of N, M and K. For example, increasing K in some cases (e.g. $N = 4, M = 10$) increased the relative performance of DBA with respect to the GH, but not in others (e.g. $N = 2, M = 10$).

Generally speaking, as the MRGAP instances become larger, the assignments found by DBA become slightly worse with respect to the GH. However, throughout the experiments, it was found that DBA is always at least competitive with the GH. When taking the best assignment matrix from 12 parallel models, DBA usually outperforms the GH across all combinations of N, M and K except when $N = 8, M = 40$, and $K \leq 2$. It is possible that with further hyperparameter tuning and training, DBA may be able to outperform the GH for any size MRGAP. Again, no additional hyperparameter tuning has been carried out since Chapter 6, so it is possible that the current hyperparameters are not well suited to large MRGAP instances.

The results demonstrate that, if DBA is to be used for solving a BAP, it is essential to understand the distribution from which it is expected that real-world problem instances will arise. If

TABLE 7.1: DBA results over the MRGAP test dataset. Note that all values are given as percentages. Therefore, the value 1 corresponds with 1% (not 100%).

N	M	K	Average over models				Best-of 12 Models			
			Mean $\overline{og}$	>GH	=GH	<GH	Mean $\overline{og}$	>GH	=GH	<GH
2	10	1	0.330	29.42	35.51	35.07	1.636	41.30	45.20	13.50
		2	-0.462	23.28	34.74	41.98	1.311	34.60	45.60	19.80
		3	-1.378	20.97	32.05	46.98	1.077	31.60	48.10	20.30
	20	1	0.738	49.42	12.76	37.82	1.519	63.10	18.70	18.20
		2	0.051	43.82	9.43	46.76	1.683	62.50	16.70	20.80
		3	-1.058	33.25	7.04	59.71	1.213	57.40	13.30	29.30
	40	1	0.831	63.80	2.72	33.48	1.461	80.40	4.10	15.50
		2	-0.346	45.98	0.96	53.07	1.350	77.70	1.70	20.60
		3	-1.290	33.64	0.92	65.44	1.039	68.20	1.70	30.10
4	10	1	0.511	28.30	38.98	32.72	1.063	36.50	55.10	8.40
		2	1.489	49.04	24.42	26.55	2.392	60.30	32.60	7.10
		3	1.914	55.78	17.37	26.85	3.333	68.80	24.20	7.00
	20	1	-0.062	19.84	29.98	50.18	0.296	31.10	57.10	11.80
		2	0.187	35.53	18.43	46.03	0.762	52.60	33.40	14.00
		3	0.544	50.34	10.69	38.97	1.342	68.90	19.30	11.80
	40	1	-0.134	11.94	20.51	67.55	0.074	23.50	57.30	19.20
		2	-0.258	18.62	11.48	69.90	0.165	39.50	34.30	26.20
		3	-0.172	27.71	8.98	63.31	0.292	51.60	26.30	22.10
8	10	1	-0.310	7.82	38.09	54.09	0.127	13.20	78.90	7.90
		2	-0.377	13.83	28.85	57.32	0.280	24.20	67.20	8.60
		3	-0.258	20.59	27.05	52.36	0.486	32.60	58.00	9.40
	20	1	-0.168	2.33	37.12	60.55	0.014	4.70	89.60	5.70
		2	-0.209	4.27	26.77	68.97	0.029	9.40	75.60	15.00
		3	-0.192	7.58	29.96	62.46	0.051	15.00	73.70	11.30
	40	1	-0.096	0.29	46.44	53.27	-0.001	0.90	97.30	1.80
		2	-0.104	0.89	23.33	75.78	-0.003	2.30	83.20	14.50
		3	-0.136	1.18	23.57	75.26	0.003	3.90	93.30	2.80

an accurate, well-constrained distribution of test instances can be derived, then that it is much easier to automate the necessary hyperparameter sweeping required to convincingly outperform more traditional approaches on real-world problem instances.

7.4.3 Runtime

For each combination of N, M and K, the query times were recorded over the test dataset. The results are presented in Table 7.2. DBA scaled roughly linearly with increasing M, while holding relatively constant for increasing N and K.

The GH was faster than DBA on average across all problem sizes. However, the GH tended to scale super linearly with increasing M and K. The GH became particularly slow as K increased if N and M were already large. It is conjectured that, as MRGAP instances become even larger (especially as K increases), DBA will eventually be faster than the GH.

DBA is relatively stable in terms of runtime, where the GH is more unpredictable. In the worst-case, the GH is in fact slower than DBA (see for example, $M = 40$, $K > 1$). Therefore, DBA can provide competitive results while also being faster in certain cases.

Summary

In this chapter, DBA was adapted to the MRGAP. It was demonstrated that DBA is competitive with a high-quality baseline. In terms of runtime, DBA was slower than the baseline, but appears to have better asymptotic growth. In addition, the baseline required expert human-knowledge and a high performance branch and bound, where DBA only required knowledge of the MRGAP description.

TABLE 7.2: DBA runtime over the MRGAP test dataset. All values are given as milliseconds. s.d. corresponds with standard deviation.

N	M	K	DBA				GH			
			Mean	Min	Max	s.d.	Mean	Min	Max	s.d.
2	10	1	12.336	7.135	16.107	1.917	0.274	0.089	1.263	0.119
		2	10.506	5.614	14.027	1.626	0.266	0.101	2.655	0.129
		3	9.545	5.898	14.481	1.774	0.260	0.107	1.139	0.103
	20	1	26.415	18.218	31.768	3.117	0.866	0.168	4.221	0.351
		2	23.458	17.826	29.557	2.658	0.515	0.149	2.636	0.236
		3	21.789	13.086	29.094	3.008	0.560	0.174	4.891	0.289
	40	1	54.584	41.858	64.069	4.919	1.378	0.358	7.495	0.731
		2	46.547	36.301	59.163	5.636	8.406	0.299	391.335	25.511
		3	44.706	32.837	54.550	4.437	14.311	0.367	470.992	38.694
4	10	1	14.590	13.184	16.079	0.396	0.788	0.295	2.506	0.224
		2	15.504	13.746	17.612	0.800	1.838	0.315	14.246	1.552
		3	14.911	10.331	17.671	1.024	0.796	0.287	3.237	0.282
	20	1	29.565	28.372	33.787	0.805	2.417	0.652	9.561	2.023
		2	29.608	27.728	31.968	0.936	2.488	0.656	7.261	1.274
		3	29.982	27.391	32.596	1.152	2.598	0.877	16.574	1.429
	40	1	63.833	60.575	67.849	1.807	3.277	1.247	9.808	1.544
		2	63.845	61.046	69.632	1.631	9.597	1.371	578.842	28.462
		3	63.145	60.210	67.925	1.790	22.856	1.813	652.132	55.206
8	10	1	15.300	14.260	17.824	0.509	1.632	0.918	4.859	0.373
		2	15.547	14.586	16.884	0.449	1.878	0.935	5.453	0.511
		3	15.491	14.628	16.774	0.429	1.877	0.837	41.002	1.641
	20	1	31.116	29.495	36.374	1.050	3.310	1.507	8.986	1.143
		2	31.469	29.832	33.820	0.850	3.396	1.889	7.056	0.547
		3	31.344	29.684	34.459	0.772	3.730	1.900	27.449	1.259
	40	1	63.213	59.898	68.276	1.784	4.798	2.433	6.959	0.458
		2	65.490	61.350	97.715	4.981	14.072	3.050	607.329	34.310
		3	64.129	60.770	70.321	1.731	25.558	4.173	497.458	43.145

Chapter 8

Conclusions

This thesis presented Deep Bipartite Assignments (DBA), a customisable deep learning architecture for automatically finding high-quality heuristics for bipartite assignment problems (BAPs). DBA is designed in particular for problems that cannot be addressed by traditional techniques: such as those that have a computationally expensive objective function and/or require high dimensional representation. In this thesis, DBA was shown to be competitive with strong baselines on two NP-Hard problems.

DBA is a general-purpose approach for heuristically solving a large class of assignment problems. This generality brings about both strengths and limitations.

8.1 Strengths

DBA requires minimal human knowledge to generate high quality assignments for arbitrary BAPs. In order to achieve optimal performance on a given BAP, hyperparameter sweeping may be required. However, this process can be easily automated with a simple script. In Chapter 7, it was also shown that DBA is capable of producing good performance without any additional hyperparameter tuning. In all of the experiments, the runtime of DBA at test time is strongly polynomial. As an example, over the multi-resource generalised assignment problem (MRGAP), DBA scaled roughly linearly with increasing M.

8.2 Limitations

Deep learning approaches are usually unable to provide any theoretical guarantees in terms of achievable objective. Where the greedy heuristic (GH) of 7.2 is able to guarantee an assignment matrix that is at least half as good as the optimal assignment matrix, DBA can only give empirical evidence of quality when compared alongside an existing method. Even when presented with empirical evidence, it is generally considered impossible to provide non-trivial guarantees about how DBA will perform on unseen BAP instances.

For very large problem instances, significant training time may be required. Depending on the exact BAP, DBA may discover strategies that are pseudo-invariant with respect to problem size. For example, when presented with the weapon-target assignment (WTA) problem, DBA can find high quality assignment matrices for larger problem instances than those seen in training. However, if this pseudo-invariant quality does not hold (as in the MRGAP), it is essential that DBA be trained on a distribution that is as close to what is expected at test time. Such a distribution may involve a large number of tasks and agents. As the number of agents and tasks grows, DBA takes significantly longer to train. When compounded by a hyperparameter sweep, deploying DBA may be infeasible in some scenarios unless high-performance resources (such as an HPC or similar) are available for training.

8.3 Future Work

In this work, DBA was validated on BAPs that already have strong existing baselines. In both cases, insufficient evidence was provided to justify that DBA should be adopted in place of these baselines. The construction heuristic of Ahuja et al., 2003 almost always outperforms DBA. In the MRGAP, the GH of Cohen et al., 2006 produced similar quality assignment matrices to DBA, but was much faster on average to compute. Therefore, perhaps the most pressing area of future research is to find an application of DBA that unambiguously outperforms all existing baselines. This thesis takes inspiration from domains such as Go (Silver et al., 2017) and Starcraft 2 (DeepMind, 2018) where deep learning easily surpasses traditional human-derived strategies. This thesis conjectures that there are BAPs where DBA reliably outperforms all existing human-derived heuristics - but this if left to future work.

Another area of future work lies in the development of communication layers. This thesis presented two communication layers that have good empirical backing: pooling and attention. In general, it was noticed that attention as a communication layer is more stable during training but is slower to both train and query. The choice between pooling and attention is a matter of user preference and the exact BAP under consideration. However, there may be other communication layers that are more effective. This thesis recommends an in-depth review of the field of geometric deep learning (Bronstein et al., 2017) to find other possible communication layers.

Bibliography

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X., and Brain, G. (2016). "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation*. Savannah, GA, USA, pp. 265–283.

Ahuja, R. K., Kumar, A., Jha, K. C., and Orlin, J. B. (2003). "Exact and Heuristic Algorithms for the Weapon-Target Assignment Problem". In: *Operations Research* 55.6, pp. 1136–1146.

Bahdanau, D., Cho, K., and Bengio, Y. (2015). "Neural Machine Translation by Jointly Learning to Align and Translate". In: *3rd International Conference on Learning Representations*. San Diego, CA, USA, pp. 1–15.

Battaglia, P. W., Pascanu, R., Lai, M., Rezende, D., and Kavukcuoglu, K. (2016). "Interaction Networks for Learning about Objects, Relations and Physics". In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. Barcelona, Spain, pp. 4509–4517.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. (2016). "Neural Combinatorial Optimization with Reinforcement Learning". In: arXiv: `1611.09940`.

Biscani, F., Izzo, D., and Yam, C. H. (2010). "A Global Optimisation Toolbox for Massively Parallel Engineering Optimisation". In: arXiv: `1004.3824`.

Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., and Vandergheynst, P. (2017). "Geometric Deep Learning: Going Beyond Euclidean Data". In: *IEEE Signal Processing Magazine* 34.4, pp. 18–42.

Cheng, J., Dong, L., and Lapata, M. (2016). "Long Short-Term Memory-Networks for Machine Reading". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, TX, USA, pp. 551–561.

Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. (2015). "The Loss Surfaces of Multilayer Networks". In: *Journal of Machine Learning Research* 38, pp. 192–204.

Cohen, R., Katzir, L., and Raz, D. (2006). "An Efficient Approximation for the Generalized Assignment Problem". In: *Information Processing Letters* 100.4, pp. 162–166.

Dai, H., Khalil, E. B., Zhang, Y., Dilkina, B., and Song, L. (2017). "Learning Combinatorial Optimization Algorithms over Graphs". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Long Beach, CA, USA, pp. 6351–6361.

DeepMind (2018). "AlphaStar: Mastering the Real-Time Strategy Game StarCraft II". In: *Blog Post*.

Defferrard, M., Bresson, X., and Pierre, V. (2016). “Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering”. In: *30th Conference on Neural Information Processing Systems*. Barcelona, Spain, pp. 3844–3852.

Deudon, M., Cournut, P., Lacoste, A., Adulyasak, Y., and Rousseau, L.-M. (2018). “Learning Heuristics for the TSP by Policy Gradient”. In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Delft, The Netherlands, pp. 170–181.

Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y. (2017). “OpenAI Baselines”. In: *GitHub Repository*.

Eiben, A. and Smith, J. (2015). *Introduction to Evolutionary Computing*. 2nd ed. Leiden, The Netherlands: Springer.

Emami, P., Pardalos, P. M., Elefteriadou, L., and Ranka, S. (2018). “Machine Learning Methods for Solving Assignment Problems in Multi-Target Tracking”. In: arXiv: `1802.06897v1`.

Foerster, J. N., Assael, Y. M., Freitas, N. de, and Whiteson, S. (2016). “Learning to Communicate with Deep Multi-Agent Reinforcement Learning”. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. Barcelona, Spain, pp. 2145–2153.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. (2017). “Neural Message Passing for Quantum Chemistry”. In: *Proceedings of the 34th International Conference on Machine Learning*. Sydney, NSW, Australia, pp. 1263–1272.

Gori, M., Monfardini, G., and Scarselli, F. (2005). “A New Model for Learning in Graph Domains”. In: *Proceedings of the International Joint Conference on Neural Networks*. Montréal, Canada, pp. 729–734.

Greensmith, E., Bartlett, P., and Baxter, J (2004). “Variance Reduction Techniques for Gradient Estimates in Reinforcement Learning”. In: *The Journal of Machine Learning Research* 5, pp. 1471–1530.

Gu, S., Lillicrap, T., Ghahramani, Z., Turner, R. E., and Levine, S. (2017). “Q-Prop: Sample-Efficient Policy Gradient with an Off-Policy Critic”. In: *5th International Conference on Learning Representations*. Toulon, France, pp. 1–13.

Guttenberg, N., Virgo, N., Witkowski, O., Aoki, H., and Kanai, R. (2016). “Permutation-Equivariant Neural Networks Applied to Dynamics Prediction”. In: arXiv: `1612.04530`.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). “Deep Residual Learning for Image Recognition”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition*. Las Vegas Valley, NV, USA, pp. 770–778.

Hochreiter, S. and Schmidhuber, J. (1997). “Long Short-Term Memory”. In: *Neural Computation* 9.8, pp. 1735–1780.

Hornik, K., Stinchcombe, M., and White, H. (1990). “Universal Approximation of an Unknown Mapping and Its Derivatives Using Multilayer Feedforward Networks”. In: *Neural Networks* 3, pp. 551–560.

Hoshen, Y. (2017). “VAIN: Attentional Multi-agent Predictive Modeling”. In: *31st Conference on Neural Information Processing Systems*. Long Beach, CA, USA, pp. 2698–2708.

Howard, R. A. (1960). *Dynamic Programming and Markov Processes*. New York, New York, USA and London, United Kingdom: Mit Press and John Wiley & Sons Ltd., pp. 32–44.

Ioffe, S. and Szegedy, C. (2015). “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Lille, France, pp. 448–456.

Jonker, R. and Volgenant, A. (1987). “A Shortest Augmenting Path Algorithm for Dense and Sparse Linear Assignment Problems”. In: *Computing* 38.4, pp. 325–340.

Kearns, M and Singh, S (2000). “Bias-Variance Error Bounds for Temporal Difference Updates”. In: *Proceedings of the 13th Annual Conference on Computational Learning Theory*. Stanford, CA, USA, pp. 142–147.

Kingma, D. P. and Ba, J. (2015). “Adam: A Method for Stochastic Optimization”. In: *3rd International Conference for Learning Representations*. San Diego, CA, USA, pp. 1–15.

Kool, W., Hoof, H. van, and Welling, M. (2019). “Attention, Learn to Solve Routing Problems!” In: *7th International Conference on Learning Representations*. New Orleans, USA, pp. 1–25.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances In Neural Information Processing Systems*. Vol. 25. Lake Tahoe, NV, USA, pp. 1097–1105.

Kuhn, H. W. (1955). “The Hungarian Method for the Assignment Problem”. In: *Naval Research Logistics* 2.1, pp. 83–98.

Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. (2016). “Gated Graph Sequence Neural Networks”. In: *4th International Conference for Learning Representations*. San Juan, Puerto Rico, pp. 1–20.

Lloyd, S. P. and Witsenhausen, H. S. (1986). “Weapons Allocation is NP-Complete”. In: *Proceedings of the 1986 Summer Computer Simulation Conference*. Reno, NV, USA, pp. 1054–1058.

Manne, A. S. (1958). “A Target-Assignment Problem”. In: *Operations Research* 6.3, pp. 346–351.

Martello, S. and Toth, P. (1990). *Knapsack Problems*. 1st ed. Chichester, United Kingdom: John Wiley & Sons Ltd.

Milan, A., Rezatofighi, S. H., Garg, R., Dick, A., and Reid, I. (2017). “Data-Driven Approximations to NP-Hard Problems”. In: *Thirty-First AAAI Conference on Artificial Intelligence*. San Francisco, CA, USA, pp. 1453–1459.

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. (2017). “Device Placement Optimization with Reinforcement Learning”. In: *Proceedings of the 34th International Conference on Machine Learning*. Sydney, NSW, Australia, pp. 1–11.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). “Human-Level Control Through Deep Reinforcement Learning”. In: *Nature* 518.7540, pp. 529–533.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). “Asynchronous Methods for Deep Reinforcement Learning”. In: *Proceedings of the 33rd International Conference on Machine Learning*. New York, New York, USA.

Öncan, T. (2007). “A Survey of the Generalized Assignment Problem and Its Applications”. In: *INFOR: Information Systems and Operational Research* 45.3, pp. 123–141.

Raina, R., Madhavan, A., and Ng, A. Y. (2009). "Large-scale Deep Unsupervised Learning using Graphics Processors". In: *Proceedings of the 26th International Conference on Machine Learning*. Montréal, Canada, pp. 873–880.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2016). "High-Dimensional Continuous Control Using Generalized Advantage Estimation". In: *4th International Conference for Learning Representations*. San Juan, Puerto Rico.

Selsam, D., Lamm, M., Bünz, B., Liang, P., Moura, L. de, and Dill, D. L. (2019). "Learning a SAT Solver from Single-Bit Supervision". In: *7th International Conference on Learning Representations*. New Orleans, USA, pp. 1–11.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). "Deterministic Policy Gradient". In: *Proceedings of the 31st International Conference on Machine Learning*. Beijing, China, pp. 387–395.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., Van Den Driessche, G., Graepel, T., and Hassabis, D. (2017). "Mastering the Game of Go Without Human Knowledge". In: *Nature* 550, pp. 354–359.

Sukhbaatar, S., Szlam, A., and Fergus, R. (2016). "Learning Multiagent Communication with Backpropagation". In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. Barcelona, Spain, pp. 2252–2260.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). "Sequence to Sequence Learning with Neural Networks". In: *Proceedings of the 27th International Conference on Neural Information Processing Systems*. Montréal, Canada, pp. 3104–3112.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. 2nd ed. Cambridge, MA, USA: MIT Press.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). "Attention Is All You Need". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Long Beach, CA, USA, pp. 6000–6010.

Vinyals, O., Fortunato, M., and Jaitly, N. (2015). "Pointer Networks". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems*. Montréal, Canada, pp. 2692–2700.

Vinyals, O., Bengio, S., and Kudlur, M. (2016). "Order Matters: Sequence To Sequence for Sets". In: *4th International Conference for Learning Representations*. San Juan, Puerto Rico, pp. 1–11.

Williams, R. J. (1992). "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Machine Learning* 8.3, pp. 229–256.

Yann, L., Leon, B., Yoshua, B., and Patrick, H. (1998). "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R., and Smola, A. (2018). "Deep Sets". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems Pages*. Long Beach, CA, USA, pp. 3394–3404.