Deep Reinforcement Learning For Assembly Line Balancing Problems

by

Wenrui Yuan

A thesis submitted in partial fulfillment for the
degree of Master of Science

in the
Faculty of Science
School of Mathematics and Statistics
THE UNIVERSITY OF MELBOURNE

May 2022

THE UNIVERSITY OF MELBOURNE

Abstract

Faculty of Science
School of Mathematics and Statistics

Master of Science

by Wenrui Yuan

With the advancements in deep reinforcement learning (RL), recent literature has explored the application of RL to problem-agnostic solution search schemes and revealed interesting insights into constructive heuristic designs for combinatorial optimisation problems.

In this thesis, we study the simple assembly line balancing problem (SALBP) with an MDP reformulation, which is formalised by the sequential decision theory. We show that this reformulation can also be generalised to a broader class of assembly line balancing problems (ALBPs). Moreover, we propose a constructive heuristic for the SALBP-1 with the help of deep learning models, whose efficiency is examined on benchmarking instances through numerical experiments. Results are then analysed and new research avenues in ALBPs are suggested.

Acknowledgements

This thesis concludes a year and half of research at the University of Melbourne. I would like to first express my deepest gratitude to my supervisor A/Prof. Alysson M. Costa for his guidance throughout this research. I appreciate the amount of time and energy he invested in exploring topics that are foreign to both of us, as well as in editing this thesis.

I would also like to thank the Master of Science (Mathematics and Statistics) course coordinator A/Prof. Jennifer Flegg for her support in planning subjects. Switching specialisation was never an easy decision for me, and I would like offer special thanks to the former coordinator A/Prof. Diarmuid Crowley for his suggestions in selecting the right pathway that helps my research.

Finally, I would like to thank my family. The past few months have been a great mental challenge impeding my study. I would not be able to complete this research without their helps.

Contents

List of Figures

List of Tables

Abbreviations

ALBP	Assembly Line Balancing Problem
ANN	Artificial Neural Network
COP	Combinatorial Optimisation Problem
DL	Deep Learning
DP	Dynamic Programming
DQN	Deep Q Network
DNN	Deep Neural Network
DRL	Deep Reinforcement Learning
GAT	Graph Attention Network
GCN	Graph Convolutional Network
GNN	Graph Neural Network
MDP	Markov Decision Process
ML	Machine Learning
PPO	Proximal Policy Optimisation
RL	Reinforcement Learning
SALBP	Simple Assembly Line Balancing Problem
SGD	Stochastic Gradient Descent
TD	Temporal Difference
TRPO	Trust Region Optimisation

Introduction

A pitcher knows the exact angle and speed for a perfect hit; a professional chess player can predict movements few rounds before their opponents; a racing driver knows when to shift gear to save fuels. The theory of decision making is so simple yet complex; it takes years of training for one to make even sub-optimal decisions but can hardly tell the reason behind them.

The idea of decision making lies at the very essence of human cognitive process: neurons accept experience and are trained to respond to a given set of stimuli. But how can one decode these neurons specific tied to decision making? And if we do, how do we read from these neurons and develop a systematic interpretation of the mechanism behind decision making?

Although we cannot answer these questions explicitly, there have been many studies on leveraging mathematical models in decision making problems. And algorithmic advancements in the past few decades have enabled us to study these problems on a numerical level through techniques like mathematical optimisation and decision trees. But still, how to establish an unified framework of methodologies for decision problems remains an interesting topic for research.

Recent advancements in deep learning has made it possible for researchers to revisit a classical decision model, the sequential decision making model, where uncertainty is taken into account and processed as state changes (system evolution); and the decision problem is solved sequentially with respect to state changes arising from internal and external uncertainties. The motivation of this

thesis is therefore to explore the feasibility of incorporating a sequential decision making framework into the design of constructive heuristics for a family of combinatorial optimisation problems, assembly line balancing problems.

Structure for the thesis is as follows: Chapter 1-4 will focus on the theoretical background for the sequential decision problem, machine learning, reinforcement learning and deep learning as well as assembly line balancing problems. Chapter 5 will explore an application of reinforcement learning to the simple assembly line balancing problem through a hybrid deep learning model, which is examined by a set of numerical experiments, investigating both generalisability and optimality of the proposed method. The final chapter will cover an overview and a discussion on further research topics that can be taken from this thesis.

Chapter 1

Sequential Decision Making

1.1 Decision Making

The presence of uncertainty has made it difficult to make decisions. It is generally impossible to provide an exact statement of why some decisions are better than others, not to mention a universal evaluation model for decision simply never exists. In general, uncertainty can arise from the environment of interest. For example, if we want to buy a lottery, it is certain that some combinations of numbers lead to the jackpot. But it is unlikely that we know the exact one, and even if we ask all lottery buyers their combinations, there are still cases where it cannot be covered. In these cases, we hold only partial information about the lottery system and the probability of picking the right combination is therefore minuscule. However, the above example can also be used to interpret the inability to predict the future, which is another cause of uncertainty in decision making. In fact, it is nearly impossible to write down a set of predefined rules about uncertainty as no outcomes in the given statement is guaranteed to occur.

Kochenderfer (2015) suggested that the decision making problem can be formalised with two components, an *agent* who makes decisions and an *environment*, which can respond to changes and holds certain physical attributes. More specifically, let the *state*, s, be the set of information which the agent can retrieve from the environment; let the *action*, a, be one or more valid decisions which the agent can take to interact with the environment. The interaction between the

agent and the environment can be viewed as the process of an agent deciding which action to take by directly observing the environment. A simple illustration of this interaction can be found in Figure 1.1, which is sometimes called the *agent-environment interface* and inherently admits the structure of dynamical systems (Richard S Sutton and Andrew G Barto 2018).

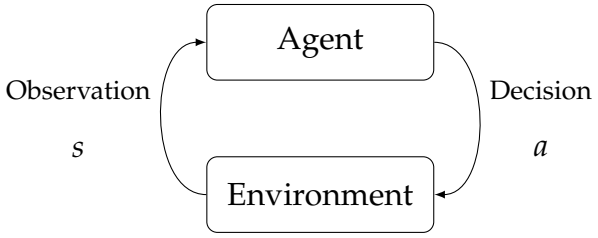


FIGURE 1.1: Interactions between agent and environment

1.1.1 Decision horizon

A decision *time horizon*, or simply as a *horizon*, is a given time periods on which our dynamical system is dependent. A horizon can be either finite or infinite, but agent is not allowed to make decisions or take actions over beyond the horizon in neither case. We will assume that, for the rest of this thesis, all dynamical systems will be determined on a finite horizon.

Formally, the horizon is defined $\mathcal{T} = 0, 1, \cdots, T-1$ for discrete systems and $\mathcal{T} = [0, T)$ for continuous ones. Let S and A be the state and action space that defines the domain of our problem. For the discrete system, one can take full exploits of the subscripts by writing $s_{t+1} = f_t(s_t, a_t)$ where $s_t, s_{t+1} \in S, a_t \in A$ and f_t denotes the system evolution (Bertsekas 2007). For the continuous systems, one can first discretise continuous-time state action space through methods like moving average.

1.1.2 Stochastic models

Although this thesis is not to review the probability theory, stochastic models will play an important role towards establishing the sequential decision making framework. In particular, we are interested in the discrete time *Markov process*, or Markov chain $\langle S, P \rangle$, where

- S is a set of states
- P is a probability distribution defined by

$$\begin{aligned} P(s_{t+1} \mid s_t) &= \mathbb{P}(S_{t+1} = s_{t+1} \mid S_t = s_t) \\ &= \mathbb{P}(S_{t+1} = s' \mid S_0 = s_0, S_1 = s_1, \cdots, S_t = s_t), \end{aligned} \tag{1.1}$$

which represents the memoryless property of Markov process. In other words, P is not a history dependent probability distribution and only depends on the current state $S_t = s_t$.

1.2 Sequential Decision Model

Animals' decision making models are rather simple: behaviours are strengthened if animals received positive stimuli from the environment they have interacted with, where the strengthening process can be thought as more frequent occurrences of such behaviours in future. The process of receiving feedbacks that encourage or discourage certain behaviours is called reinforcement. To put it simple, the animal learning model contains 4 major variables, action, state, *policy* and *reward* (Andreae 1969).

1.2.1 Definition

The simplest reward-control model defines:

- the reward function $R : S \times A \to \mathbb{R}$, which computes the immediate reward of making action a at state s; and
- the transition function T.

In particular, the reward can be either a positive one that encourage certain actions, or a negative one that hinders some actions to be performed by the agent. And the transition can be modeled with deterministic function of the form $T(s, a) = s'$, or a probability distribution $T(s, a) = P(s' \mid s, a)$.

If the transition is modeled as a stochastic one, the probability distribution can be either history dependent, meaning that

$$P(s' \mid s, a) = \mathbb{P}(S_{t+1} = s' \mid S_0, A_0, S_1, \cdots, S_t, A_t); \tag{1.2}$$

or independent of history $H = (0, 1, \cdots, t)$:

$$\begin{aligned} P(s' \mid s, a) &= \mathbb{P}(S_{t+1} = s' \mid S_t = s, A_t = a) \\ &= \mathbb{P}(S_{t+1} = s' \mid S_0, A_0, S_1, \cdots, S_t, A_t), \end{aligned} \tag{1.3}$$

which satisfies the Markov property (1.1).

The agent also needs to follow certain formats of decision rule called policy, which is often denoted by π that can be either deterministic or stochastic. In particular, if

- π is deterministic, then $\pi(s_t) = a_t$ at any given time step t; and
- if π is stochastic, $\pi(s) = \pi(a, s) = \mathbb{P}(A_t = a \mid S_t = s)$ is a probability distribution.

Decision rules are sometimes used interchangeably with deterministic policies, which can be *stationary* or *non-stationary*, namely $\pi(s_t) = d$ or $\pi(s_t) = d_t$ for

all $t \in \mathcal{T}$. For stochastic policy, recall that the sum of all probabilities of all the events in a sample space is equal to 1, any stochastic policy must therefore follow

$$\sum_{a \in A} \pi(a \mid s) = 1, \tag{1.4}$$

for any state $s \in S$.

1.3 Markov Decision Process

So far we have established a set of rules and variables for building a sequential decision making model. However, as discussed earlier, uncertainty can arise from the model itself or from decision-induced state changes that effect model dynamics. The change in model dynamics is generally driven by internal or external stochasticity; internal stochasticity arises from the environment or system of interest, and is usually dependent on time. External stochasticity comes the agent or the decision maker. Moreover, the reward system can be seen as another level uncertainty which affects the agent's decision.

Although it can be sometimes convenient to review historical events over the decision horizon, it is computationally inefficient in general and hard to distinguish state dependence (i.e. some states are dependent on others). Therefore, we can recall the Markov model in (1.1). Define a new transition function with the Markov process by letting R_{t+1} be the a random variable of representing the reward received at time t; given that action A_t is performed, the state transition function in (1.3) can be reformulated as:

$$P(s', r \mid s, a) = \mathbb{P}(S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a) \tag{1.5}$$

1.3.1 Definition

A *Markov decision process* (MDP) is a tuple $\langle S, A, P, R \rangle$ (Bellman 1957; Howard 1960; Watkins 1989), where:

- S is the state space
- A is the action space that can be dependent or independent of states
- P is the transition probability, defined by

$$\begin{aligned} P(s' \mid s, a) &= \mathbb{P}(S_{t+1} = s' \mid S_t = s, A_t = a) \\ &= \sum_{r \in R} P(s', r \mid s, a). \end{aligned} \tag{1.6}$$

- R is the expected reward that can be defined by either

$$\begin{aligned} R(s, a) &= \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a] \\ &= \sum_{r \in R} rP(s' \mid s, a), \end{aligned} \tag{1.7}$$

  or

$$\begin{aligned} R(s, a, s') &= \mathbb{E}[R_{t+1} \mid S_{t+1} = s', S_t = s, A_t = a] \\ &= \sum_{r \in R} \frac{rP(s', r \mid s, a)}{P(s' \mid s, a)}. \end{aligned} \tag{1.8}$$

All of the above definitions need to follow the Markov property as discussed in Subsection 1.1.2. In particular, (1.6) can be seen as sampling over all possibles reward of (1.5), which can be useful if we are only interested in states and actions, but not in the stochasticity of reward received. Meanwhile, the difference between (1.7) and (1.8) is that the former does not have information about the new state s' and can be thought as the reward being state independent and

only dependent on the action performed. Both sets of definitions are used interchangeably in the reinforcement learning community (Richard S Sutton and Andrew G Barto 2018; Watkins 1989), but (1.7) will be preferred for the rest of this thesis.

1.3.2 Optimality condition

In real world practice, decisions can have both short and long term effects, so a decision maker should be aware what is the short and long term return of performing certain actions. Andrew Gehret Barto, Richard S Sutton, and Watkins (1989) addressed this concern by introducing a discounted cumulative sum:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{1.9}$$

where $\gamma \in (0, 1]$ is a discount factor. Intuitively, this is to estimate the discounted sum of all future reward starting from time step t, which identifies long term rewards (Watkins 1989).

We can then establish the connection between states and rewards to, in some sense, evaluate states. In particular, the expected overall return for any state $s \in S$ under policy π and over a finite horizon $\mathcal{T} = \{t, \cdots, T\}$ is often called a *state value function* (Richard S Sutton and Andrew G Barto 2018; Watkins 1989), defined by:

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{T-1} \gamma^k R_{t+k+1} \mid S_t = s \right] \end{aligned} \tag{1.10}$$

Similarly, one can define the *state-action value function* by adding stochasticity arisen from the action space:

$$\begin{aligned} Q_\pi(s,a) &= \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{T-1} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right], \end{aligned} \tag{1.11}$$

which associate rewards with states and actions. Using condition (1.4) and observe (1.11) and (1.10) to conclude that

$$V_\pi(s) = \sum_{a \in A} \pi(a \mid s) Q_\pi(s,a), \tag{1.12}$$

which reveals hidden assumption about $V_\pi(s)$ that, if one policy π' is no worse than another, say π, then $V_{\pi'}(s) \geq V_\pi(s)$ must hold. Now, write down the Bellman equation of $V_\pi(s)$:

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi\left[\sum_{k=0}^{T-1} \gamma^k R_{t+k+1} \mid S_t = s\right] \\ &= \mathbb{E}_\pi\left[\gamma^0 R_{t+1} + \gamma \sum_{k=0}^{T-1} \gamma^k R_{t+k+2} \mid S_{t+1} = s\right] \\ &= \sum_{s' \in S} P_\pi(s' \mid s,a)\left[R_\pi(s,a,s') + \gamma \mathbb{E}_\pi\left[\sum_{k=0}^{T-1} \gamma^k R_{t+k+2} \mid S_{t+1} = s'\right]\right] \\ &= \sum_{s' \in S} P_\pi(s' \mid s,a)\left(R_\pi(s,a,s') + \gamma V_\pi(s')\right) \\ &= \sum_{a \in A} \pi(a \mid s)\left(\sum_{s' \in S} P(s' \mid s,a)\left(R(s,a,s') + \gamma V_\pi(s')\right)\right) \\ &= \sum_{a \in A} \pi(a \mid s)\left(R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) V_\pi(s')\right), \end{aligned} \tag{1.13}$$

which, according to (1.12), implies that

$$\begin{aligned} Q_\pi(s,a) &= R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a') V_\pi(s') \\ &= R(s,a) + \gamma \sum_{s' \in S} \sum_{a' \in A} P(s' \mid s,a') \pi(a' \mid s') Q_\pi(s',a') \end{aligned} \tag{1.14}$$

To find the optimal solution to MDP at any state $s \in S$ is therefore to maximise (1.10) and (1.11):

$$V^*(s) = \max_\pi V_\pi(s) \quad \text{and} \quad Q^*(s,a) = \max_\pi Q_\pi(s). \tag{1.15}$$

In particular, one has

$$\begin{aligned} Q^*(s,a) &= \max_\pi Q_\pi(s) \\ &= R(s,a) + \gamma \max_{a' \in A} \sum_{s' \in S} P(s' \mid s,a') Q^*(s',a') \end{aligned} \tag{1.16}$$

and

$$\begin{aligned} V^*(s) &= \max_{a \in A} Q^*(s,a) \\ &= \max_{a \in A} \mathbb{E}\left[R_{t+1} + \gamma V^*(S_{t+1}) \mid S_t = s, A_t = a \right]. \end{aligned} \tag{1.17}$$

Chapter 2

Reinforcement Learning

2.1 Machine learning fundamentals

2.1.1 Introduction

So far we have established the environment through stochastic decision model but not the agent. In general, to make a decision, an agent can either be informed by a certain set of rules or it can *learn* the decision. As discussed earlier, it is often very challenging to script a set of decision rules for a stochastic dynamical system, and would require expert knowledge to the problem of interest; and hence comes the idea of learning.

The term *learning* studied by the engineering discipline often refers to a set of tools called *machine learning* (ML), seen as a subset of *artificial intelligence* (AI). Although there is no common agreement on the exact definition on learning, the mechanism of ML algorithms can be interpreted by learning from *experience* or *examples*, which is usually described by a large amount of similar data for a specific task (Mitchell 1997; Carbonell et al. 1983).

Although often being used interchangeably with the term deep learning (DL), ML is not necessarily the same as a DL model. Instead, ML refers to three main paradigms of algorithms:

- Supervised learning, which takes input and output pairs to predict output labels of unseen input, guided by external supervisions from the performance measure.
- Unsupervised learning, which takes only the input data and predict labels without external supervision.
- Reinforcement learning, which requires a reformulation of the problem to a sequential decision making one.

Supervised learning algorithms concern mainly regression and classification tasks, whereas unsupervised learning algorithms are mostly seen in clustering analysis and anomaly detection (G. Hinton and Sejnowski 1999; Mohri, Rostamizadeh, and Talwalkar 2018), Reinforcement learning, on the other hand, can be applied to a broader range of problems from control theory (Gullapalli 1992), operations research and combinatorial optimisation (Hubbs et al. 2020; Mazyavkina et al. 2021), game theory (Nowé, Vrancx, and Hauwere 2012) and much more.

Modern reinforcement learning (RL) developments, however, depend on supervised learning methods as the approximation scheme provided by supervised learning usually delivers faster convergence and less memory spaces (Richard S Sutton and Andrew G Barto 2018; Mnih, Kavukcuoglu, et al. 2013). As such, although this section is not to provide an overview on the supervised learning, some fundamentals do benefit later formalisation of deep reinforcement learning algorithms.

2.1.2 Supervised learning

In general, a supervised learning algorithm takes n independent and identically distributed (i.i.d) data points $\{(x^i, y^i)\}_{i=1}^n$, or a *batch* (Goodfellow, Yoshua Bengio, and Courville 2016), from a probability distribution P_{data} over sample

space $X \times Y$. The goal is to approximate a predictor function (or a hypothesis) $h : X \to Y$ such that $h(x) = \hat{y}$. To evaluate the approximation, define the loss function, or the prediction error $\ell : Y \times Y \to \mathbb{R}^+$ such that $\ell(h(x), y)$ returns the difference between predicted results $\hat{y}$ and true results y. Hence the objective of supervised learning is to minimise the risk:

$$R(h^*) = \min_{h \in \mathcal{H}} \mathbb{E}[\ell(h(x)), y], \tag{2.1}$$

while the optimal hypothesis is:

$$h^* = \arg\min_{h \in \mathcal{H}} \mathbb{E}[\ell(h(x)), y]. \tag{2.2}$$

The difference between an optimisation task and a supervised learning one is that, the true distribution $P_{data}(x, y)$ is not known and one needs to consider the *data-generating distribution*, or *empirical distribution*, from the data set (Goodfellow, Yoshua Bengio, and Courville 2016). The goal of a supervised learning task is therefore to minimise the empirical risk:

$$\mathbb{E}[\ell(h(x)), y] = \frac{1}{n} \sum_{i=1}^{n} \ell(h(x^i), y^i). \tag{2.3}$$

Above methods render impractical as the explicit form of the hypothesis function is usually not of interest. Instead, h can be approximated through a set of parameters θ. Write $h_\theta(x) = h(x; \theta)$, one can derive a cost function

$$J(\theta) = \mathbb{E}[\ell(h(x; \theta)), y]. \tag{2.4}$$

And (2.1) therefore reads

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell(h(x^i; \theta) y^i). \tag{2.5}$$

2.1.3 Gradient descent

It is well known that optimisation of differentiable functions can be solved by the gradient descent (GD) method. The loss function is in general assumed to be differentiable and convex, the iterative update therefore reads:

$$\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta), \tag{2.6}$$

where θ is the parameterisation of the hypothesis and α denotes the *learning rate* that controls the rate of convergence and solution quality: a learning rate too high results in early convergence to suboptimal solutions, whereas low learning rate yields slow convergence and risks of converging to local minima (maxima) (Ruder 2016). Non-differentiable loss functions are rarely seen in modern ML practice as ML methods rely heavily on the gradient updates.

However, stochastic models suffer greatly from the curse of dimesionality and performing GD on a full batch of inputs can induce significant computational stress and unnecessary memory usage. Instead, one can define the sample estimates:

$$g = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \ell(h(x^i; \theta), y^i), \tag{2.7}$$

and apply GD update with g:

$$\theta \leftarrow \theta - \alpha g, \tag{2.8}$$

with $m \leq n$. This method, inspired by the Monte-Carlo method, is called *mini-batch gradient descent* or *stochastic gradient descent* (SGD) (Goodfellow, Yoshua Bengio, and Courville 2016). Setting m sufficiently large provides a good approximation to the true loss and the true gradients.

2.1.4 Generalisation

For ML problems, the term *generalisation* specifically refers to the transferability from one dataset to the other. More specifically The set of data used to construct the model is often called a *training set*, while the dataset used to validate the model is called *testing set* (Goodfellow, Yoshua Bengio, and Courville 2016). Borrowed from statistical inference model (Kenneth P. Burnham 2002), ML literature tends to use *underfitting* or *overfitting* to describe types of *error* encountered in generalisation (Bousquet and Elisseeff 2000). In particular, both error occur when the model yields poor performance on the testing test while underfitting occurs when the model fails to bound prediction loss even within desired range even on the training set.

2.2 Reinforcement learning

The learning aspect of RL is less obvious compared to supervised learning as there is no output y and the idea of "label" does not exists in the pure RL framework. Rather, the objective of RL is to maximise the expected return by iteratively updating the return until convergence. Meanwhile, definitions in RL can vary from one study to another due to its widespread application and theoretical backgrounds. Unless specified, RL in this section refers to the general framework for algorithms to solve sequential decision making problems embedded with fully observable MDPs.

A *trajectory* is often defined as a set of state-action pairs over a horizon $\mathcal{T} = 0, 1, \cdots$ (i.e. $(s_0, a_0, s_1, a_1, \cdots)$), such that if the horizon has finite dimension, the corresponding problem is termed *episodic* and the complete roll-out of states $\{s_0, s_1, \cdots, s_t\}$ is called an *episode*. Subsets of an episode is often called *experience* or sometimes as batch, identifying a partial trajectory $(s_i, a_i, \cdots, s_k, a_k)$ for $\{i, \cdots, k\} \subset \mathcal{T}$ (Lange, Gabel, and Riedmiller 2012), often in a sequential

order. This should not to be confused with the idea of batch in other ML literature, where a batch generally implies a set of often non-sequential sub-tasks that can well represent the data for generalisation (Goodfellow, Yoshua Bengio, and Courville 2016). To avoid confusion, some RL literature refers to word "instance" when it comes to generalisation (Kirk et al. 2021).

2.2.1 Model-free reinforcement Learning

If the system dynamic is fully known, then (1.17) has an iterative solution (Ross 1983)

$$\begin{aligned} V_{k+1}(s) &= \mathbb{E}_\pi \left[R_{t+1} + \gamma V_k(S_{t+1}) \mid S_t = s \right] \\ &= \sum_{a \in A} \pi(a \mid s) \sum_{s' \in S} P(s' \mid s, a)(R(s', a) + \gamma V_k(s')), \end{aligned} \tag{2.9}$$

which is often called the *iterative policy evaluation* in dynamic programming (DP) studies (Richard S Sutton and Andrew G Barto 2018).

Here it branches out two subcategories of RL: *model-based* and *model-free*. Model based, as the name suggests, requires a model for the agent to learn. In this case, we are interested in the transition probability $P(s' \mid s, a)$ that, by tracing the state-action trajectories $s_0, a_0, s_1, \cdots, s_t$, leads to the learning of transition function $T(s, a) = s'$ (Moerland, Broekens, and Jonker 2020). On the other hand, the model-free method will rely on finding the policy function $\pi(a \mid s)$ to maximise the state value function. DP and later formalisation of stochastic DP solutions provides a theoretical background for the model-free method (Bertsekas 2007; Ross 1983).

2.2.2 Value and policy iteration

In Subsection 1.3.2, we have assumed that for any $\pi' \geq \pi$, $V_{\pi'}(s) \geq V_\pi(s)$ must hold. This is certainly true if we assume $\pi(s) \neq \pi'(s) = a$ and define

$\pi'(s) \geq \pi(s)$, which leads to $Q_{\pi'}(s, \pi'(s)) = V_{\pi'}(s)$ from (1.12). Optimal policy can therefore be found with

$$\begin{aligned} \pi^*(s) &= \arg\max_{a \in A} Q_\pi(s, a) \\ &= \arg\max_{a \in A} \sum_{s' \in S} P(s' \mid s, a)(R(s, a, s') + \gamma V(s')). \end{aligned} \tag{2.10}$$

This greedy search scheme for optimal policy which rolls out a best actions a, depending on an old state s and a new state s', is called a *policy iteration* method, which perform a one step lookahead computation at any state. The idea of this iterative procedure is to continuously evaluate policy through $V(s)$ and improve the policy by greedily select action a that yields highest $V(s)$ (Richard S Sutton and Andrew G Barto 2018). As shown in Algorithm A.2, policy or action updates iteratively according to the evaluation from Algorithm A.1.

Clearly, the iterative search for optimal policy incurs a lot redundancy, which is in general less efficient. To avoid redundancies in the nested searching loops, we can interact directly with $V(s)$ (Richard S Sutton and Andrew G Barto 2018). According to (2.9), we can rewrite the evaluation method by

$$V_{k+1} = \max_a \sum_{s' \in S} P(s' \mid s, a)(R(s', a) + \gamma V_k(s')), \tag{2.11}$$

which yields a new searching scheme by iteratively updates $V(s)$ without directly interact with the policy. Such a method is called a *value iteration* method requires a lot less computation (see Algorithm A.3).

2.2.3 Temporal-difference learning

The amount of computation required in either iteration methods grows proportionally with the problem sizes. Hence, the general practice tends to approximate a solution with heuristics (Richard Stuart Sutton 1984; Richard S Sutton 1992; Tesauro 1992).

On the other hand, as value iteration method can start updating estimates at any given state, the amount of examples for learning is significantly reduced compared to policy iteration method. One straight-up approach is to solve (2.11) with DP. However, DP suffers greatly from extensive sizes of action space since it requires a full traverse of all possible actions with each policy update. Richard Stuart Sutton (1984) proposed a more sample-efficient method called the temporal-difference (TD) learning, in which the value updates directly by setting:

$$V(s_t) \leftarrow V(s_t) + \alpha[R_{t+1} + \gamma V(s_{t+1}) - V(s_t)], \tag{2.12}$$

where α is defined as the *learning rate* indicating the weighted return caused by state transition. This simplest form is sometimes specified as TD(0), which only requires a one-step-lookahead and no evaluation of single action. The simple read out of $R_{t+1} + \gamma V(s_{t+1})$ matches with the definition of (2.9), and update the existing estimate in a tabulation way is very similar to DP. As shown in Algorithm A.4, TD generally requires less computation than DP since DP performs full backtrack while TD only looks at the current and following state. The n-step generalisation of (2.12) is called a TD(λ) method (Richard S Sutton 1992), often paired with multi-layer neural network design (Tesauro et al. 1995).

The fast convergence delivered by TD learning relies on batch updating (Lange, Gabel, and Riedmiller 2012), a similar technique seen in the SGD method, where a mini-batch of past states (or state-action pairs) is used to perform the learning.

In this case, as long as the α in (2.12) is sufficiently small, TD(0) converges to a single solution (Richard S Sutton and Andrew G Barto 2018).

2.2.3.1 On-policy TD: SARSA

The term *on-policy* method is used specifically for algorithms that take policy into account, as opposed to the term *off-policy*, where algorithms either have no information or are indifferent about the exact action. An on-policy TD exploits $Q_\pi(s, a)$ and the STATE-ACTION-REWARD-(NEXT)STATE-(NEXT)ACTION pair, abbreviated by SARSA. After collecting enough batches SARSA pairs, the TD update reads:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]. \tag{2.13}$$

Implementation can be easily adapted from Algorithm A.4 by changing line 8 and the data structure of experience E accordingly.

2.2.3.2 Off-policy TD: Q learning

The Q learning was first introduced in Watkins (1989) as a variant of TD learning. It is also an off-policy method that computes (1.16) instead of (1.17). More specifically, the update rule in (2.12) can redefined by

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]. \tag{2.14}$$

We observe that the difference between (2.14) and (2.13) is that, (2.14) updates the Q value estimate without explicitly interacting with action a_t. On the other hand, Q learning does not require a full episode of state-action pairs, provides a much better sample efficiency than SARSA.

2.2.4 Monte-Carlo learning

The policy iteration method generally requires complete episodes to update. We naturally brings out the Monte-Carlo method to mitigate some redundant

computation and achieve faster convergence. Although it is generally not as sample efficient as the TD method (Lange, Gabel, and Riedmiller 2012; Richard S Sutton and Andrew G Barto 2018), being able to interact with policy directly may benefit some problems, especially ones that have a small action space.

An on-policy Monte-Carlo learning generally follows:

1. Initialise a random policy π and a arbitrary state value function V and empty list G'
2. Collect an complete episode of states $E = (s_0, s_1, \cdots, s_t)$ using π.
3. For each $s_i \in E$, compute $G_i = \sum_{j=i}^{T-1} \gamma^j R_{i+j+1}$ and add G_i to G'
4. $V(s_i) = \overline{G'}$ (average of return starting from s_i)
5. Read out $\pi(s_i) = \arg\max_a Q_\pi(s_i, a)$
6. Update π and repeat from step 2 until the termination condition is met.

Clearly, the major downside of original Monte-Carlo learning compared to TD method is the amount of data required grows at least qudratically with the problem size. To reduce some computation and to observe the direct update on the policy with an explicit trace of actions, collection and updates on $V(s_i)$ in the previous searching scheme can changed to $Q(s_i, a_i)$ (Tsitsiklis 2003; Watkins 1989). This revised model requires collecting complete trajectories (i.e. $E = (s_0, a_0, s_1, a_1, \cdots, s_t)$) before evaluation, a strategy called exploring starts (ES) (Richard S Sutton and Andrew G Barto 2018). In particular, it changes the single dimensional tracing G' provided by the original Monte-Carlo to a two dimensional tabulation, as shown in Algorithm A.5.

However, the lack of justification on initial policy roll out cannot guarantee convergence in some cases (Tsitsiklis 2003). For example, if the initial policy is

poorly selected, improvement on top of that introduces uncertainties; or if the initial policy is "hard", meaning that $\pi(s) = a$ almost indefinitely for any s, results in poor or no improvements. One way to offset this underfitting or overfitting dilemma is to define the "soft" policy controlled by a probability ϵ, resulting the so called *ϵ-greedy* policy (Richard S Sutton and Andrew G Barto 2018). The intuition of ϵ-greedy is to introduce a (decaying) parameter that help smooth the convergence curve. In particular, the agent has a probability of ϵ choosing a random action and a probability of $1 - \epsilon$ choosing an action according to the policy π. With (1.12) and (1.17), rewrite the optimality condition by:

$$\begin{aligned} V_*^\epsilon(s) &= (1-\epsilon)\max_a Q_*^\epsilon(s,a) + \frac{\epsilon}{|A(s)|}\sum_{a\in A} Q_*^\epsilon(s,a) \\ &= (1-\epsilon)\max_a \sum_{s'\in S} P(s' \mid s,a)[R(s,a,s') + \gamma V_*^\epsilon(s')] \\ &+ \frac{\epsilon}{|A(s)|}\sum_{a\in A(s)}\sum_{s'\in S} P(s' \mid s,a)[R(s,a,s') + \gamma V_*^\epsilon(s')], \end{aligned} \tag{2.15}$$

where $A(s)$ is the state dependent action space given state s.

Chapter 3

Deep Learning

3.1 Introduction

Deep learning (DL) refers to a subset of ML algorithms that employ artificial neural networks (ANN) as a approximation scheme for the original ML problem. The word "deep", comes from the architecture of ANN with multiple hidden layers. A hidden layer can be thought of intermediate computations performed by hidden units (neurons) between inputs and outputs (Goodfellow, Yoshua Bengio, and Courville 2016; Yann LeCun, Yoshua Bengio, and Geoffrey Hinton 2015). A layer can be any mathematical computation on a specified number of nodes, each carries a weight and a bias that can be updated through training. Modern DL algorithms are able to treat unprocessed inputs without specified format and do not require domain-specific knowledge if the given dataset contains enough features (Y. Bengio, Courville, and Vincent 2013; Yann LeCun, Yoshua Bengio, and Geoffrey Hinton 2015).

3.2 Feed-forward networks

The history of neural networks can be traced back to the 1940-1950s (Hebb 2005; Farley and Clark 1954). The perceptron model, believed by many, was the first practice of neural networks (Minsky and Papert 1969). Yet, research stagnates on a symbolic level until the introduction of backpropagtion algorithm (Werbos 1975). In particular, the backpropagtion was applied to trace loss and distribute

the loss on each hidden units for the neural network to update (Y. LeCun et al. 1989). This backpropagation process is often called *training* or a network update.

The feed-forward network (FFN), as a special type of ANN, adapts from the multi-layer perceptron (MLP) model to propagate computation in a single direction (Hornik 1991; Schmidhuber 2015), as opposed to the recurrent neural network (RNN) (Rumelhart and McClelland 1987), which feed outputs back into the model recursively. In the FFN model, given an input encoded into vector form, hidden units on each layer gather vector elements apply linear transformation so that the transformed input can be passed into the consecutive layer (Y. LeCun et al. 1989). The simplest form of network layers is the *fully-connected* (FC) layers, where the output of each layer can be defined by an Affine transformation of inputs. An example of FC layers in FFN model can be found in Figure 3.1. In particular, let $x \in \mathbb{R}^n$ be the input data and $y \in \mathbb{R}^d$ be the output; assuming weights $W \in \mathbb{R}^{n \times d}$ and bias $b \in \mathbb{R}^d$ and output can be defined by an Affine transformation:

$$y = Wx + b. \tag{3.1}$$

Then we use the backpropagtion algorithm to assign losses to each hidden units to update W and b. Recall that in Subsection 2.1.2, we use θ to parameterise the cost function for supervised learning tasks. If one assumes $h_\theta = \sum_{i=1}^{n} \theta_i x_i$, which, in matrix form, has $h_\theta(x) = \theta^T x$, it is virtually the same as the network weight W.

In general, a FFN model approximate a functions $y = f(x)$ using through hidden layers h with linear weights and bias (Goodfellow, Yoshua Bengio, and Courville 2016). While in RL research, a FFN can be used to learn either the transition $s' = T(s, a)$ or the policy rule $P(s' \mid \pi(s), s)$ that maps state-action pair to probability distribution of transitions.

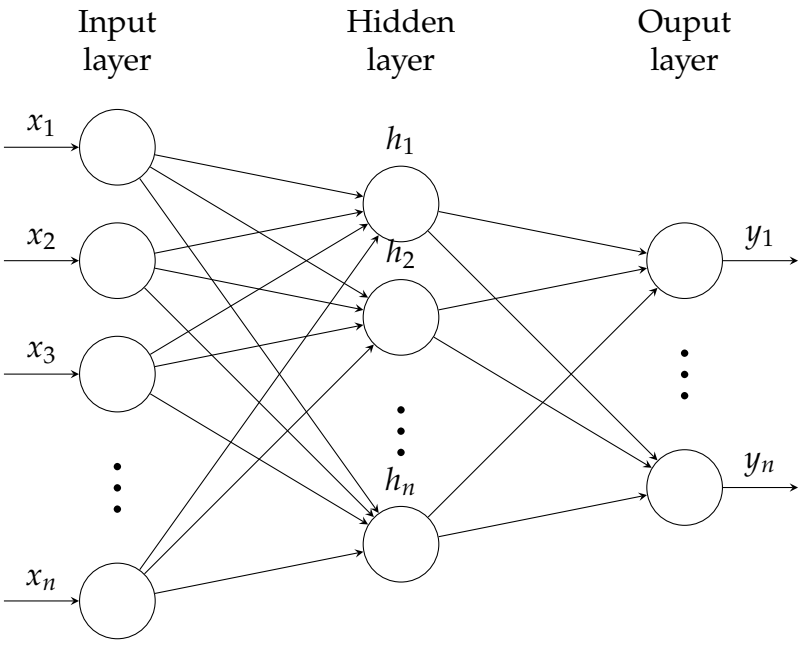


FIGURE 3.1: Illustration of a feed-forward network with fully-connected layers

3.2.1 Forward pass

As a MLP model, FFN passes input data into multiple hidden layers h in a sequential order. To understand the mechanism of *passing*, suppose each hidden layer $h^{(i)}$ carries weights W_i and b_i as defined in (3.1). Then a forward pass is simply

$$h^{(i+1)} = \sigma(W_i h^{(i)}, b_i), \tag{3.2}$$

with $x = h^{(0)}$ and $y = h^{(n)}$, assuming a n-layer network. Motivated by the neuron activation process, the function σ is an non linear activation function that applies to the all elements in the vector $h^{(i)}$. Justification for using an activation function is rather complex (Goodfellow, Yoshua Bengio, and Courville 2016), but in rare cases of vanishing gradient (Hochreiter 1991), activation function regularises output vector for weights and biases to be updated normally. Like loss functions, the purpose of activation function is widespread and can range from setting output threshold to smoothing gradients (Datta 2020). This thesis

mainly concerns three activation functions: *rectified linear unit* (ReLU), softmax function and the LeakyReLU, defined by:

$$\begin{aligned} \text{ReLU}(x) &= \max\{0, x\}, \\ \text{softmax}(x) &= \frac{e^x}{\sum_j e^{x_j}}, \\ \text{LeakyReLU}(x) &= \max\{ax, x\}, \end{aligned} \tag{3.3}$$

where a is an arbitrarily small scalar, often takes the value of 0.01 (Maas, Hannun, and Ng 2013).

3.2.2 Convolutional neural networks

A convolutional neural network (CNN) is a class of ANN that employs convolutional layers and downsampling layers to process high dimesional data. It was first developed in (Fukushima 1980) with the introduction of *C-cells* inspired by human receptive fields. Later improvement formalised the convolutional and downsampling layers and found application in hand-written digits recognition (Lecun et al. 1998). The space-invariance property was discovered in a similar network layer design (Zhang et al. 1990), levitating research potentials beyond image recognition (Grefenstette et al. 2014; Defferrard, Bresson, and Vandergheynst 2016).

In particular, gievn an image input I and kernel filter K that move over the input to collect image features, the output of a convolutional layer is exactly the discrete convolution (Goodfellow, Yoshua Bengio, and Courville 2016):

$$S_{i,j} = (I * K)_{i,j} = \sum_m \sum_n I_{m,n} K_{i-m,j-n}, \tag{3.4}$$

where $*$ is the convolution operator. Each layer can be then constructed using (3.2). However, unlike fully connected layers, convolutional layers is sparsely

connected. More specifically, local connectivity between filters and neurons to the next layers is established, a not so obvious observation from (3.4).

In general, CNN refers to the paradigm of applying convolution and down-sampling layers to inputs before passing to fully-connected layers, rather than a standalone network model. It can therefore be applied to FFN (Lecun et al. 1998), RNN (Wang et al. 2018) or any ANNs.

3.3 Graph neural networks

The graph neural network (GNN) is a family of network architecture desgined for data with graph structures. Originally presented as the message passing neural network (Scarselli et al. 2009), problem-specific developments arises naturally with the study of molecules using supervised learning (Gilmer et al. 2017) and large network problems (Veličković et al. 2017; Kipf and Welling 2016; Hamilton, Ying, and Leskovec 2017). Later studies generalise GNN applications to geometrical data has revealed several connections and research potential to embed non-Euclidean input data with DNN (Bronstein et al. 2017).

In general, GNN models assume a graph input $G = (V, E)$ with $|V| = N$ nodes in the form of:

- a matrix $X \in \mathbb{R}^{N \times F}$ representing F nodal features; and
- an adjacency (sparse) matrix $A \in \mathbb{R}^{N \times N}$ representing the graph structure.

Following (3.2), each GNN layer computes:

$$h^{(i+1)} = f(h^{(i)}, A), \tag{3.5}$$

where $h^{(0)} = X$ and $h^{(n)} = Z$ is the output. An example of GNN embedding can

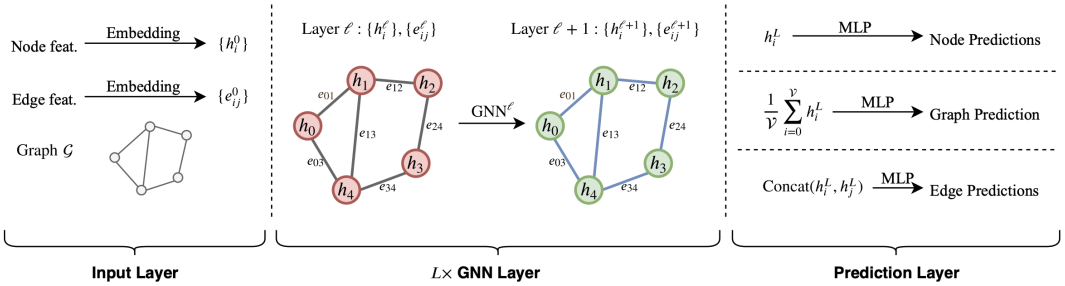


FIGURE 3.2: Standard pipeline for GNN models (Dwivedi et al. 2020)

be found in Figure 3.2.

Definitions of f vary from model to model, but a large body of literature accepts the graph convolution neural network (GCN) as the baseline model. The intuition of GCN is closely related to CNN, which also uses a convolutional filter specifically designed for graph structured data instead of images.

In particular, there are two formulations of graph (spectral) convolution based on spectral graph theory (Hammond, Vandergheynst, and Gribonval 2009). In Defferrard, Bresson, and Vandergheynst (2016), the convolution filter is a spectral filter g_θ, defined by

$$g_\theta(L)X = g_\theta(U\Lambda U^T)X = Ug_\theta(\Lambda)U^T X, \tag{3.6}$$

where

$$L = \mathbb{1}_N - D^{-1/2}AD^{-1/2} = U\Lambda U^T \tag{3.7}$$

is the normalized graph Laplacian with eigenvector U and diagonal matrix Λ of eigenvalues. To avoid extensive computation of egiendecomposition in large graphs (Hammond, Vandergheynst, and Gribonval 2009), Chebyshev polynomial of order k can well parameterise g_θ by:

$$g_\theta(\Lambda) = \sum_{k=0}^{K} \theta^k T_k(\tilde{\Lambda}), \tag{3.8}$$

where $\tilde{\Lambda} = 2\Lambda/\lambda_{\max} - \mathbb{1}_N$. This leads to a paramterisable filter:

$$Y_{n,j} = \sum_{i=1}^{F} g_{\theta_{i,j}}(L) X_{n,i}, \tag{3.9}$$

for each node $n \in N$ and the model is constructed by following standard CNN propagation.

On the other hand, Kipf and Welling (2016) proposed a 1-hop layer-wise approximation:

$$g_\theta * X \approx \theta_0 X + \theta_1 (L - \mathbb{1}_N) X = \theta_0 X - \theta_1 (D^{-1/2} A D^{-1/2}) X. \tag{3.10}$$

The propagation rule, inspired by the Weisfeiler-Lehman algorithm, is specified for each node:

$$h_u^{(i+1)} = \sigma \left(\sum_{v \in \mathcal{N}(u)} \frac{1}{c_{u,v}} h_v^{(l)} W^{(l)} \right), \tag{3.11}$$

where $\mathcal{N}(v)$ is the 1-hop neighbourhood of node v and $c_{u,v} = \sqrt{|\mathcal{N}(u)| \cdot |\mathcal{N}(v)|}$ (Kipf and Welling 2016).

However, both models only offer treatment to undirected graphs. The generalisation to all other types of graphs is soon addressed in Veličković et al. (2017) by proposing a self-attention-based GNN model with graph attentional layer (GAT). Explaining the self-attention model (Vaswani et al. 2017) involved in GAT requires a full overview of RNN models and the early developments of attention mechanisms (Bahdanau, Cho, and Y. Bengio 2014; Mnih, Heess, et al. 2014), which are not of particular interest to this thesis. In fact, Veličković et al. (2017) have explicitly defined the node level self-attention as a score between node i, j:

$$E_{u,v} = a(Wh_u, Wh_v). \tag{3.12}$$

Note that the weights $W \in \mathbb{R}^{F \times N}$ are shared across neighbouring nodes. To learn the attention function, define a learnable parameter $\mathbf{a} \in \mathbb{R}^{2F}$ with

$$E_{u,v} = \text{LeakyReLU}(\mathbf{a}^T \cdot [Wh_u \| Wh_v]), \tag{3.13}$$

where $\|$ is the concatenation of vectors (i.e. $[Wh_u, Wh_v]^T$). Hence, the normalised attention is:

$$A_{u,v} = \text{softmax}_v(e_{u,v}) = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^T \cdot [Wh_u \| Wh_v]))}{\sum_{w \in \mathcal{N}(u)} \exp(\text{LeakyReLU}(\mathbf{a}^T \cdot [Wh_u \| Wh_w]))}, \tag{3.14}$$

and the propagation rule is then defined by

$$h_u^{(i+1)} = \sigma\left(\sum_{v \in \mathcal{N}(u)} A_{u,v} W h_v^{(i)} \right), \tag{3.15}$$

a result similar to (3.11) (Veličković et al. 2017). More specifically, if there exists up to k independent attention heads, each with individual weights and attention scores, the propagation rule becomes (Vaswani et al. 2017; Veličković et al. 2017)

$$h_u^{(i+1)} = \sigma\left(\frac{1}{K} \sum_{k=0}^{K} \sum_{v \in \mathcal{N}(u)} A_{u,v}^K W^K h_v^{(i)} \right). \tag{3.16}$$

3.4 Deep Reinforcement Learning

Deep reinforcement learning (DRL) refers to algorithms that solve RL problems with the help deep neural networks (DNNs). Traditional RL algorithms suffer greatly from the expensive tracing of value functions or policy update, while DRL provides a much faster convergence and stable results by incorporating SGD methods (Mnih, Kavukcuoglu, et al. 2013; Richard S Sutton, McAllester, et al. 1999).

3.4.1 Off-policy method

The Deep Q Network (DQN) is perhaps the most iconic off-policy DRL algorithm in modern RL studies (Mnih, Kavukcuoglu, et al. 2013). DQN is established on the work done in Watkins (1989), which performs TD updates on the Q value. The intuition of DQN is to construct a DNN to update Q value instead of the standard TD tabulation. As formalised in Subsection 2.1.2, the cost function with respect to the Q value can parameterised by:

$$J(\theta_i) = \mathbb{E}_{s,a\sim P(s,a)}\left[(y_i - Q(s,a;\theta_i))^2\right], \tag{3.17}$$

where

$$y_i = \mathbb{E}_{s'\sim P(s'|s,a)}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) \mid s,a\right] \tag{3.18}$$

denotes the *target* and $P(s,a)$ is called the behaviour distribution, a distribution of possible state-action pairs. The gradient loss then follows

$$\nabla_{\theta_i} J(\theta_i) = \mathbb{E}_{P(s'|s,a)}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]. \tag{3.19}$$

One can then perform the SGD with as in (2.7) to minimise the loss.

3.4.2 On-policy method

Like DQN, we can also approximate the policy update with a parameter θ. Formally, let $\tau = (s_0, a_0, s_1, \cdots, s_T, a_T)$ be trajectories rolled out by policy π_θ, the parameterisation of policy reads:

$$\pi_\theta(\tau) = P(s_0)\prod_{t=1}^{T}\pi_\theta(a_t \mid s_t)P(s_{t+1}, r_{t+1} \mid s_t, a_t), \tag{3.20}$$

where $P(s_0)$ marks the probability of yielding initial state s_0. Moreover, write the cost function

$$J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)]. \tag{3.21}$$

then the gradient of policy update is defined by (Richard S Sutton, McAllester, et al. 1999):

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\gamma^t R(s_t, a_t)\sum_{j=0}^{t}\nabla_\theta \log \pi_\theta(a_j \mid s_j)\right]. \tag{3.22}$$

Derivations of are due to Richard S Sutton, McAllester, et al. (1999) and not to our particular interest. The difference between this gradient based policy update method and is that, we want to maximise (3.21) in this case. That is, we want the gradient ascent $\theta \leftarrow \theta + \alpha\nabla_\theta J(\pi_\theta)$.

Chapter 4

Assembly Line Balancing Problem

4.1 Introduction

Assembly lines are mass production systems that divide the manufacturing of product into indivisible *tasks*. Tasks have their own *processing time* and must be assigned to *workstations*, where they are completed within a *cycle time*, beyond which no task is allowed to be performed on that station. Due to technological restrictions, tasks are also imposed with precedence constraints, which is usually a partial ordering that specifies which tasks must come before others. More specifically, a task can only be assigned to a station after all of its predecessor tasks have already been assigned to the same or earlier stations.

4.2 The simple assembly line balancing problem

The simple assembly line balancing problem (SALBP) concerns the assignment of tasks to stations while respecting the precedence relations. The problem is called type-1 (SALBP-1) if the objective is to minimise number of stations with a given cycle time; and is called type-2 (SALBP-2) if the number of stations is fixed and the objective is minimise the cycle time. In addition, a type-E SALBP (SALBP-E) optimises the line capacity by minimising both the cycle time and the number of stations.

Exact and heuristic methods for both SALBP-1 and SALBP-2 have been extensively studied in Gunther, Johnson, and Peterson (1983), Baybars (1986), Talbot, Patterson, and Gehrlein (1986), Scholl and Voß (1997), and A. Otto and C. Otto (2014) for its importance in studying the general assembly line balancing problem (GALBP) (Becker and Scholl 2006).

4.2.1 Integer programming for SALBP-1

A standard notation for SALBP can be found in Table 4.1. An instance of SALBP-

Notation	Definition
n	the number of tasks
$m(\overline{m})$	(upper bound on) the number of stations
C	cycle time
$N = \{1, 2, \cdots, n\}$	set of tasks
$M = \{1, 2, \cdots, m\}$	set of stations
$F_i(F_i^*)$	set of immediate (all) followers of task $i \in N$
$P_i(P_i^*)$	set of immediate (all) predecessors of task $i \in N$
p_i	processing time of task $i \in N$
$e_i = \lceil (p_i \sum_{k \in P_i^*} p_k)/C \rceil$	earliest station of task $i \in N$
$l_i = \overline{m} - \lceil (p_i \sum_{k \in F_i^*} p_k)/C$	latest station of task $i \in N$
$sl_i = l_i - e_i$	slack time of task $i \in N$

TABLE 4.1: Notations of SALBP

1, in particular, contains a precedence relation usually in the form of graph $G = (N, E)$, with $N = \{1, 2, \cdots, n\}$ and $E = \{(i, j) \mid i \in N, j \in F_i\}$. The upper bound on the number of stations $\overline{m}$ is rarely given but one can either consider the trivial bound $\overline{m} = n$ or a theoretical upper bound (Scholl and Voß 1997), defined by:

$$\overline{m} := \min \left\{ n, \left\lfloor \frac{\sum_{i \in N} p_i}{C + 1 - \max_j p_j} \right\rfloor + 1, \left\lfloor \frac{2 \sum_{i \in N} p_i}{C + 1} \right\rfloor + 1, \right\}. \tag{4.1}$$

Different integer programming formulations of SALBP-1 exist according to the selection of decision variables (Baybars 1986). In particular, the most common form of integer programming formulation uses impulse variables (Ritt and Costa 2018):

$$x_{ji} = \begin{cases} 1 & \text{if } i \in N \text{ is assigned to } j \in M \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

The integer program can then be formulated by

$$\begin{aligned}
\min \quad & \sum_{j \in M} y_j && && \text{(4.3a)} \\
\text{s.t.} \quad & \sum_{i \in N} p_i x_{ji} \leq C y_j && \forall\, j \in M && \text{(4.3b)} \\
& \sum_{j \in M} x_{ji} = 1 && \forall\, i \in N && \text{(4.3c)} \\
& x_{kl} \leq \sum_{j \in M | j \leq k} x_{ji} && \forall\, i \in N, l \in F_i, k \in S && \text{(4.3d)} \\
& y_j \in \{0,1\} && \forall\, j \in M && \text{(4.3e)} \\
& x_{ji} \in \{0,1\} && \forall\, i \in N, j \in M && \text{(4.3f)}
\end{aligned}$$

In particular, (4.3d) respects the precedence constraints. (4.3b) ensures cycle limit is not exceeded for each station. And (4.3c) ensures each task is assigned only once.

4.2.2 Priority rule method

A priority rule method (PRM) for SALBP-1 solves the problem by sequentially assigning tasks to stations, subject to their priority values (Talbot, Patterson, and Gehrlein 1986). Depends on the rule, priorities of tasks must be considered in increasing or decreasing order of priority values. If the selected task results in

the violation of (4.3b), the current station is closed and the task is assigned to the next station instead (Talbot, Patterson, and Gehrlein 1986).

There are generally two types of priority rules: elementary and composite (A. Otto and C. Otto 2014). Elementary rules are often one-to-one mappings from tasks to single values. They are well investigated and classified in Talbot, Patterson, and Gehrlein (1986) and Scholl and Voß (1997). Composite rules are weighted sum of different priority rules or sometimes the aggregation of elementary rules (A. Otto and C. Otto 2014).

Following the notation in Table 4.1, some elementary rules can be explicitly computed as shown in Table 4.2: We identify PRM as a greedy search procedure that

Elementary rules		**Formulation** pv_i
Max. Task Time	(MaxT)	p_i
Max. time divided by latest station	(MaxTdL)	t_i/l_i
Max. time divided by slack	(MaxTdS)	t_i/sl_i
Min. latest station	(MinL)	l_i^1
Max. positional weights	(MaxPw)	$p_i + \sum_{k\in F_i^*} p_k$
Max. number of all followers	(MaxF)	$\lvert F_i^*\rvert$

[1] Here is to minimise l_i

TABLE 4.2: Selected elementary priority rules for SALBP-1

can be reformulated by MDP with deterministic policy. More specifically, the policy can be defined as any of the priority rules listed in Table 4.2 and the MDP searches for actions based locally optimal decisions at each time step t. Formally, let priority value of task i be pv_i for each $i \in N$ and suppose valid action space $A(s_t) \subseteq N$ denotes the set of tasks available to be assigned. Then the optimal decision $a_t \in A(s_t)$ under deterministic policy d at any given state s_t can be found with

$$d(s_t) = a_t := \underset{i\in N\setminus\{a_0,a_1,\cdots a_{t-1}\}}{\arg\max} pv_i, \tag{4.4}$$

or

$$a_t = \underset{i \in N \setminus \{a_0, a_1, \cdots a_{t-1}\}}{\arg\min} \quad pv_i, \tag{4.5}$$

if the goal is to find tasks with minimum priority values.

4.2.3 A MDP formulation

The motivation behind formulating SALBP-1 with MDP arises from the fact that PRM inherently admits a sequential decision rule as argued in (4.4). The policy generated with a PRM method is deterministic, so that the decision rule itself and the optimality condition do not change over time and the output is non-stationary and state-dependent. Of course we cannot define the state naively based on the priority values of tasks. Otherwise the agent loses temporal (time t) and spectral (graph structure) information about the problem instance and prone to violate either (4.3c) or (4.3d).

To address these concerns, we propose two kinds of formulations:

(a) **Value-based**: Consider only the value update fw_i, where:

$$fw_i^{(0)} = \sum_{j \in P_i^*} p_j, \quad \forall\, i \in N \tag{4.6}$$

represents the state at time $t = 0$. For an arbitrary state s_t, it follows that:

$$s_t = (fw_1^{(t)}, fw_2^{(t)}, \cdots, fw_n^{(t)}) \tag{4.7}$$

The state-dependent action space is defined as

$$A(s_t) = \{i \mid i \in N, fw_i^{(t)} = 0\}. \tag{4.8}$$

Moreover, we update s_t by

$$s_{t+1} = (fw_1^{(t+1)}, fw_2^{(t+1)}, \cdots, fw_n^{(t+1)}), \tag{4.9}$$

where, suppose an action a_t indicates task i being selected,

$$fw_i^{(t+1)} = -1, \quad \text{and} \quad fw_j^{(t+1)} = \begin{cases} fw_j^{(t)} - p_i & \text{if } i \in P_i^* \\ fw_j^{(t)} & \text{otherwise} \end{cases} \quad \forall\, j \neq i \tag{4.10}$$

Insight of this method comes from the ranked positional weight (RPW) priority rule as listed in Table 4.2.

(b) **Graph-based**: Perform explicit updates on the precedence graph:

$$\begin{aligned} s_0 &= (N, E) \qquad \text{and} \\ s_t &= (N_t, E_t), \quad \text{for } t = 1, \cdots T \end{aligned} \tag{4.11}$$

The action space is defined as

$$A(s_t) = \{i \mid i \in N_t, (j,i) \notin E_t \quad \forall\, j \in N_t\}. \tag{4.12}$$

Then, $a_t = \pi(s_t)$ leads to the state transition

$$s_{t+1} = (N_t \setminus \{a_t\}, E_t \setminus \{(a_t, j) \mid j \in F_{a_t}\}), \tag{4.13}$$

where F_{a_t} is the set of immediate followers of $a_t \in A(s_t) \subset N$.

Definition of the reward function concerns two aspects: immediate reward and long-term reward. The immediate reward is a positive impulse:

$$R(a_t, s_t) = p_{a_t} \cdot |F_{a_t}|, \tag{4.14}$$

which takes both processing time and the number of followers into account. Our motivation here comes from the two priority rules MaxT and MaxF as listed in Table 4.2. As the reward is imminent, or in some sense local, the use if immediate successors, $|F_i|$, is preferred over the use of all successors, $|F_i^*|$.

The long-term reward follows the subgoal shaping as introduced in Laud (2004). In particular, the subgoal is to minimise the amount of idle time on each station. Formally, suppose the the current station $m \in M$ is closed at time t (i.e. no further tasks can be selected without violating (4.3b)). Let t_{idle} be the final idle time of current station m, then the reward is given by:

$$\tilde{R}(a_t, s_t) = R(a_t, s_t) - \beta \cdot t_{\text{idle}}, \tag{4.15}$$

where β is a scaling factor. This "bonus" reward penalises the agent for leaving too much idle time, incentivising it to achieve local optimality on the current station m. Moreover, after all tasks are assigned, a final reward is given to the agent. In particular, let $\tilde{m}$ be the number of stations constructed by RL procedure, the final reward is defined by

$$R(s_{T-1}, a_{T-1}) = -\delta \cdot \tilde{m}, \tag{4.16}$$

where δ is another scaling factor controlling the penalisation of objective function.

4.3 Extensions to other ALBPs

Solution procedure for SALBP can be indeed extended to GALBP (Becker and Scholl 2006). In particular, the formulation in Subsection 4.2.3 naturally accepts many other ALBPs.

UALBP-1

The major difference between the type-1 U-line assembly line balancing problem (UALBP-1) and SALBP-1 is the line layout, where the U-shaped assembly line design allows workers to move across workstations and thus producing a more flexible assignment. In the example of Figure 4.1, we can either assign 4 workers to stations that are front and back to workers, grouped by red dotted lines, or suggest a different layout that uses only 3 workers by having two of them working on three adjacent stations (to their front, back and right or left), grouped by blue dashed lines. The MDP formalised in Subsection 4.2.3 can be therefore

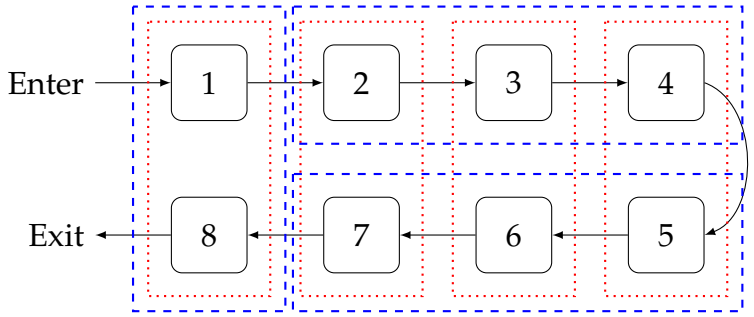


FIGURE 4.1: Two configurations (in red dotted and blue dashed lines) of U-shaped assembly line

applies to UALBP-1 without any issues. The final configuration is processed by searching adjacent stations with excessively large idle time and then reschedule tasks across these stations and group them into a single station. In this case, the MDP formulation for SALBP-1 does not necessarily solve the UALBP-1 and a final reconfiguration of SALBP-1 solution is required.

Another possible formulation uses a predefined configuration of workstations. That is, we can treat adjacent stations as a single station, with varying capacity depends on their original cycle time. For example, in Figure 4.1, if station 2, 3 and 4 each have a cycle limit of C, then the grouped configuration of $\{2, 3, 4\}$ has a total capacity of $3C$. Then, simply solve the SALBP-1 assignment, where t_{idle}

as defined in (4.15) now depends on the station configuration $m \in M$ (e.g. keeps a predefined list of station capacity $C_1, C_2, \cdots, C_{m'}$, where m' is the number of stations in the predefined configuration).

SALBP-E

The extension to SALBP-E can be formulated by controlling the scaling factor in (4.15) and (4.16) since the objective of SALBP-E is to minimise the line capacity $T = m \cdot C$ (i.e. the number of stations and cycle time). For our model to adapt SALBP-E, we can consider gradually shrinking the cycle time C, until the original assignment requires a new station.

The SALBP-E is rarely studied (Esmaeilbeigi, Naderi, and Charkhgard 2015), and therefore, the extension of our proposed method would requires the definition of appropriated priority rules. We suggest an alternative DRL practice, which is to formulate another MDP for SALBP-2 and construct a dual RL agent to solve SALBP-2 with respect to the agent on SALBP-1. The action space will depend on both agents and requires a multi-agent RL environment (Hoen et al. 2006).

SUALBSP

The setup assembly line balancing and scheduling problem (SUALBSP) is yet another extension to SALBP-1 where the original graph $G = (N, E)$ is associated with a weight function $w : F \to \mathbb{R}$ (Andrés, Miralles, and Pastor 2008), with E^* being the transitive closure of E, establishing possible edges between all nodes. The setup time between task i and j is defined by $ts_{i,j} = w(i, j) \geq 0$ for any $(i, j) \in E^*$. An example instance of SUALBSP can be found in Figure 4.2.

To adapt our developments to the SUALBSP, we can use the same graph-based method with a slight modification in the reward function. Formally, if a set of actions $(a_{t_i}, \cdots, a_{t_j-1})$ is performed before a_{t_j}, the the immediate reward at time

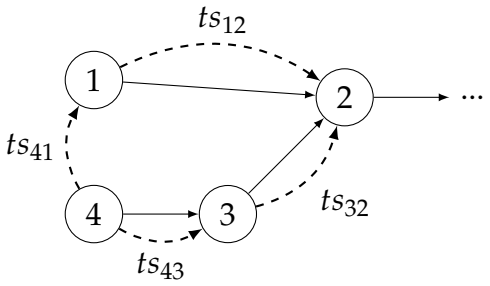


FIGURE 4.2: Example instance of SUALBSP, dashed line represents sequence dependent setup time

t_j is defined as:

$$\tilde{R}(s_{t_j}, a_{t_j}) = R(s_{t_j}, a_{t_j}) + \sum_{t_k \in \{t_i, \cdots, t_j - 1\}} ts_{a_{t_k}, a_{t_j}}, \tag{4.17}$$

which essentially adds setup time to the processing time. The action space definition remains the same as (4.12). State transition, on the other hand, is rather complicated. A feasible way is to update the transitive graph E^*:

$$E^*_{t+1} = E^*_t \setminus \{(a_t, j) \mid ts_{a_t, j} \neq 0\} \cup \{(j, a_t) \mid ts_{j, a_t} \neq 0\}. \tag{4.18}$$

The state transition is then defined the similar way as in (4.13) by adding an extra element E^*_{t+1} (i.e. $s_{t+1} = (N_{t+1}, E_{t+1}, E^*_{t+1})$).

Chapter 5

Application

5.1 Methodology

Solving an SALBP-1 instance requires problem-specific knowledge and sometimes instance-specific implementation that is hard to transfer to other ALBPs. We identified this gap in research, and propose a problem-agnostic implementation of SALBP-1 that only requires input data through deep reinforcement learning (DRL). The proposed DRL model for the SALBP-1 can be found in Figure 5.1. Note that blue dashed lines indicate the training (i.e. network update) process, without which will be a random search procedure.

In particular, we suggest a PPO+GAT model as introduced in Sections 3.3 and 3.4. The PPO agent is responsible for solving the MDP by reading out neural network outputs from the embedding of GAT. The benefit of using GAT, and inherently a graph-based MDP formulation is that GAT embeds the graph topology into a single level node (task) evaluation and it is therefore less subjective to instance scaling.

In particular the precedence graph is encoded with an adjacency matrix A, along with node-level features

$$X = [p_1, p_2, \cdots, p_n]^T, \tag{5.1}$$

where p_i is the processing time of task $i \in N$ specified in the input data. Then, at each time step t, the tuple (A, X) is passed into GAT, which outputs a vector

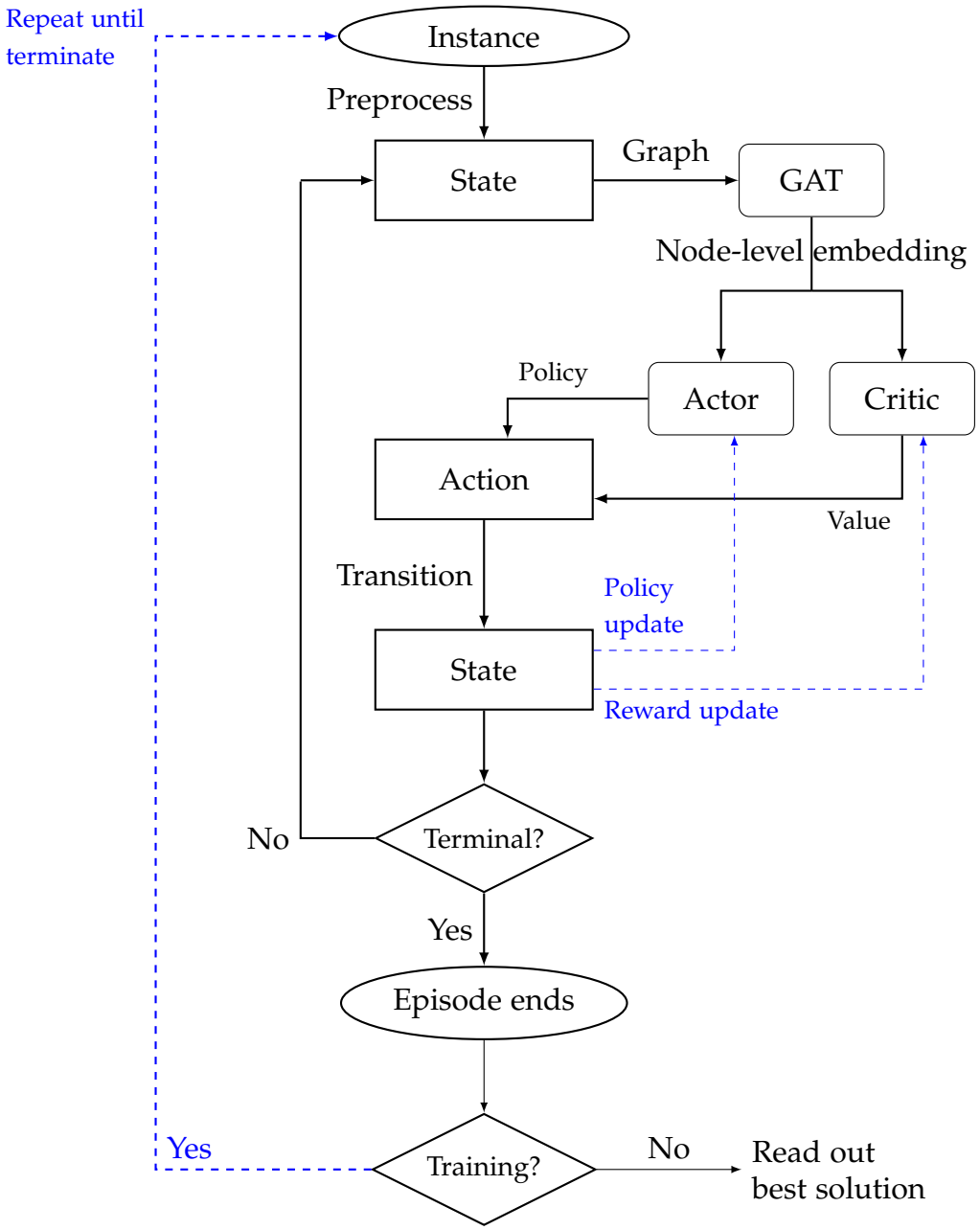


FIGURE 5.1: DRL model for SALBP-1; blue dashed lines indicate the training procedure

$Z = [l_1, l_2, \cdots, l_n]$ where each l_i can be seen as a score on task i. The actor-critic network will then take GAT outputs and learn theses weights through another 2 fully-connected layers.

Actions are not always valid because the neural network is agnostic of constraints (4.3b), (4.3c) and (4.3d). There are several ways to address this problem but we will be focusing on a strategy called invalid action masking proposed in Huang and Ontañón (2020). Suppose the network output at each state s_t is given by $Z = [l_1, l_2, \cdots, l_n]$, the first level of masking is defined as an element-wise operator, $M_1(\cdot)$, such that

$$M_1(l_i) = \begin{cases} l_i & \text{if } i \in A(s_t) \\ L & \text{otherwise} \end{cases}, \tag{5.2}$$

where $L < 0$ is set to be sufficiently small so that $\text{softmax}(L) \approx 0$. This takes care of constraints (4.3c) and (4.3d). Ensuring (4.3b) will require a second level of masking:

$$M_2(l_i) = \begin{cases} l_i & \text{if } i \in S_m + p_i \leq C \\ L & \text{otherwise,} \end{cases} \tag{5.3}$$

where S_m is the station load of current station m. To incorporate both constraints, consider a masking composition:

$$M_1 \circ M_2(l_i) = \begin{cases} M_1(l_i) & \text{if } M_2(M_1(l_i)) = L \quad \forall\, i \in N \\ M_2(M_1(l_i)) & \text{otherwise} \end{cases}, \tag{5.4}$$

where this masking composition ensures that the selected task will always respect precedence constraints. The cycle time constraints might be violated, which, in that case, implies that a new station must be opened.

To show how this masking works, consider a simple SALBP-1 instance of size 4, with only one precedence $(1, 3)$ as shown in Figure 5.2. Further assume the processing time is given by $[600, 470, 500, 319]$ and the cycle time is capped at $C = 1000$.

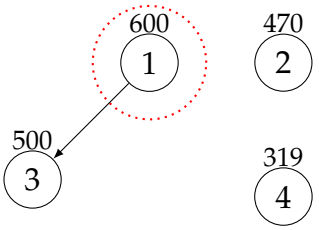


FIGURE 5.2: Example SALBP instance

Now, if task 1 is selected, then $M_1(Z) = [L, l_2, l_3, l_4]$ and $M_2(M_1(Z)) = [L, L, L, l_4]$. The probability of selecting task 4 is

$$\text{softmax}(l_4) = \frac{\exp(l_4)}{3\exp(L) + \exp(l_4)}, \tag{5.5}$$

which virtually goes to 1 as $\exp(L) \to 0$.

5.2 Numerical Experiments

5.2.1 Experimental setup

We perform numerical experiments to explore the DRL formulation in two aspects, namely:

- Correlation between cumulative reward and solution quality
- Generalisation to other instances

All experiments are implemented with TensorFlow and performed on a NVIDIA GeForce RTX 3060 GPU. For the general deep learning setup, each instance is trained for 100 episodes with a learning epoch of 10 for every 5 steps of observations. For the PPO agent, the policy clip rate is set to 0.2 with no epsilon greedy involved. Discount factor γ is set to 0.99 for numeric stability with a general advantage estimate decay of 0.95. The GAT-embedded policy network

uses a single head attention across 2 hidden layers with 32 hidden units, while the value network parameters and losses are set to defaults as in Schulman et al. (2017). Other parameters, including learning rate, optimiser and weight initialiser, are the same as TensorFlow default settings.

5.2.2 Benchmarking instances

Instances used for the following experiments are selected from the dataset as presented in A. Otto, C. Otto, and Scholl (2013). The original dataset contains 525 instances each of size 20, 50, 100 and 1000, and are classified by a "trickiness" measure based on the gap between the heuristic solution and the known integer programming minimum or lower bound. Moreover, instances can be classified into 3 categories according to the distribution of processing times in the input data. For the testing part, most instances in the benchmarking dataset are selected, with some being omitted due to triviality in their results.

5.2.3 Results

Training

For each problem size ($n = 20, 50$ and 100), we trained the agent sequentially on 5 instances that are classified as *extremely tricky* for 100 episodes. Regularising factors β and σ, defined in (4.15) and (4.16), are set to vary with instance size, but with an overall scale of 10.

Training results can be found in Figure 5.3, where each subfigure on the left column (i.e. Figures 5.3a, 5.3c and 5.3e) is cumulative rewards during the entire training process. And each subfigure on the right column (i.e. Figures 5.3b, 5.3d and 5.3f) represents actual solutions over 100 episodes. In particular, both cumulative rewards and solutions obtained during training are compared with those obtained from random search heuristic and the PRM method using MaxTdL.

Testing

The trained DRL model is then tested on selected instances and solutions are compared with other heuristic solutions. In particular, optimality gaps between heuristic solutions and known minima, as well as the percentage of solutions reaching known optima are measured to evaluate these heuristics. Instances are also classified into two main categories, trickiness and time distribution. For the trickiness measure, we are interested in *extremely tricky*, *very tricky* and *tricky* instances. And for the time distribution, there are three sub-categories of concern, namely the *peak in the middle*, *peak at the bottom* and *bimodal* [1]. In particular, for larger instances (i.e. $n = 50, 100$), there are some with no known minimum. Such instances are classified as open and included in our testing results. Results for instances of size $n = 20, 50$ and 100 can be found in Table 5.1, Table 5.2 and Table 5.3.

5.2.4 Discussion

As shown in Figure 5.3, solutions obtained from our DRL procedure exhibit an early-stage convergence and stay in stable range in later stages of training. It is also obvious that improvements in cumulative reward correlate to improvements in solution quality, which implies that our reward shaping indeed helps regulate the agent to search for better actions. Both trends are less obvious for instances of smaller sizes (i.e. $n = 20, 50$), where we argue that

- the gap between a random search solution and a heuristic solution for a small instance is inherently small, which implies that there are essentially less space for the DRL agent to improve the solution; and

[1] The original dataset contains another category based on graph structure of the instance, which is omitted in our results as their differences in solution qualities are rather trivial

		$n = 20$						
Method	**Category**	**Ours**	**MinL**	**MaxF**	**MaxPw**	**MaxT**	**MaxTdL**	**MaxTdS**
Gap	**Trickiness**							
	Extremely tricky	12.01%	13.48%	15.44%	13.76%	13.61%	13.02%	12.71%
	Very tricky	4.58%	14.00%	14.34%	11.90%	11.42%	9.25%	9.27%
	Tricky	2.44%	10.52%	10.60%	8.38%	6.74%	5.98%	6.22%
	Time distribution							
	Peak in the middle	4.78%	7.32%	7.77%	5.63%	4.52%	3.52%	3.56%
	Bimodal	2.99%	6.31%	6.20%	5.51%	3.52%	3.00%	2.88%
	Peak at the bottom	6.38%	11.81%	12.19%	11.00%	9.05%	8.76%	8.95%
	Overall	4.65%	8.48%	8.72%	7.38%	5.70%	5.09%	5.13%
Optim. Found*	**Trickiness**							
	Extremely tricky	7.41%	0.00%	11.11%	11.11%	3.70%	7.41%	11.11%
	Very tricky	23.94%	11.27%	11.27%	22.54%	33.80%	45.07%	45.07%
	Tricky	67.68%	42.07%	41.46%	56.71%	68.29%	74.39%	73.17%
	Time distribution							
	Peak in the middle	52.00%	31.43%	30.29%	45.71%	56.57%	65.14%	64.57%
	Bimodal	85.14%	68.57%	70.29%	74.86%	84.43%	86.29%	86.86%
	Peak at the bottom	81.14%	66.86%	69.71%	72.00%	74.86%	75.43%	74.86%
	Overall	72.62%	55.51%	56.65%	64.07%	71.48%	75.48%	75.43%

* Measured in % of solutions attained known minima within the selected category

TABLE 5.1: Results on selected SALBP-1 instances of size $n = 20$; instances are classified by trickiness and time distribution

- reward received is not strong enough for the agent to modify its current policy.

In fact, both shortcomings come from the fact that the DRL agent learned to maximise its reward solely by exploring benefits of performing certain actions. Less permutations in task selections lead to a smaller solution space, and therefore relatively sparse subgoal rewards. Recall that our final reward is received after all stations are closed and the agent is informed by the discounted future reward. Therefore, in smaller instances, the agent is aware of the final reward in the form of discounted cumulative sum of reward, because the final step is relatively closer to the agent. In this case, it is largely influenced by the final reward. That it, if the immediate reward of performing another action is not strong

		$n = 50$						
Measure	**Category**	**Ours**	**MinL**	**MaxF**	**MaxPw**	**MaxT**	**MaxTdL**	**MaxTdS**
Gap	**Trickiness**							
	Open*	11.37%	12.73%	12.98%	11.14%	9.38%	8.80%	9.22%
	Extremely tricky	8.51%	10.10%	8.36%	9.19%	7.76%	6.96%	7.18%
	Very tricky	4.58%	8.31%	7.81%	6.42%	4.78%	4.31%	4.27%
	Tricky	2.44%	5.02%	5.04%	3.20%	2.90%	1.79%	2.45%
	Time distribution							
	Peak in the middle	10.01%	10.88%	11.13%	9.48%	7.62%	6.45%	6.92%
	Bimodal	3.06%	4.82%	4.72%	3.74%	3.09%	2.87%	2.95%
	Peak at the bottom	2.91%	3.70%	4.10%	3.91%	3.14%	2.96%	3.30%
	Overall	4.70%	6.47%	6.65%	5.71%	4.62%	4.09%	4.39%
Optim. Found**	**Trickiness**							
	Open*	-	-	-	-	-	-	-
	Extremely tricky	4.27%	1.22%	0.61%	5.49%	10.37%	12.20%	10.37%
	Very tricky	52.78%	11.11%	19.44%	36.11%	50.0%	58.33%	58.33%
	Tricky	78.05%	51.22%	52.44%	75.61%	75.61%	84.15%	78.05%
	Time distribution							
	Peak in the middle	2.86%	2.29%	2.29%	5.14%	8.00%	11.43%	8.57%
	Bimodal	64.57%	42.86%	44.57%	58.29%	63.43%	66.29%	65.14%
	Peak at the bottom	80.00%	74.29%	71.43%	74.29%	78.86%	80.00%	78.29%
	Overall	49.05%	39.73%	39.35%	45.82%	50.00%	52.47%	50.57%

* Instances whose optima are not given

** Measured in % of solutions attained known minima within the selected category

TABLE 5.2: Results on selected SALBP-1 instances of size $n = 50$; instances are classified by trickiness and time distribution

enough for it explore, it will stick to the action read from the policy. Meanwhile, because our model uses the greedy policy, which only considers assignments on the current station, this further discourages the agent to explore alternative assignment if the original assignment is optimal enough.

On the other hand, we observe from Figures 5.3b and 5.3d that the DRL model converges to local optima and stops improving completely. This is often seen in RL models when the value function fails to estimate future rewards, in which case the policy cannot be improved due to the lack of incentives. To overcome local optima, one can introduce some stochasticity to the training process by either adding an entropy when updating the policy network (increased system entropy)

$n = 100$								
Measure	**Category**	**Ours**	**MinL**	**MaxF**	**MaxPw**	**MaxT**	**MaxTdL**	**MaxTdS**
Gap	**Trickiness**							
	Open*	12.47%	13.60%	13.68%	12.84%	10.39%	9.42%	9.74%
	Extremely tricky	7.18%	8.36%	8.36%	7.25%	6.23%	5.37%	5.55%
	Very tricky	3.01%	3.92%	3.07%	2.08%	1.76%	0.91%	0.75%
	Tricky	1.56%	2.45%	1.28%	0.87%	0.60%	0.17%	0.17%
	Time distribution							
	Peak in the middle	13.10%	12.49%	12.66%	11.13%	9.10%	7.74%	8.13%
	Bimodal	3.68%	4.16%	3.88%	3.13%	2.82%	2.47%	2.53%
	Peak at the bottom	2.43%	2.75%	2.27%	2.22%	1.94%	1.62%	1.53%
	Overall	5.18%	6.46%	6.27%	5.49%	4.62%	3.94%	4.06%
Optim. Found**	**Trickiness**							
	Open*	-	-	-	-	-	-	-
	Extremely tricky	5.09%	2.03%	3.73%	8.14%	10.58%	14.58%	14.58%
	Very tricky	44.19%	27.97%	39.53%	62.79%	67.44%	81.39%	83.72%
	Tricky	74%	60%	80%	86%	90%	96%	96%
	Time distribution							
	Peak in the middle	0.00%	0.00%	0.00%	0.00%	0.57%	1.71%	2.29%
	Bimodal	25.14%	16.57%	22.29%	34.86%	37.71%	44.57%	42.86%
	Peak at the bottom	66.29%	61.71%	68.57%	69.14%	73.14%	77.71%	78.86%
	Overall	30.5%	26.1%	30.3%	34.7%	37.1%	41.3%	41.3%

* Instances whose optima are not given

** Measured in % of solutions attained known minima within the selected category

TABLE 5.3: Results on selected SALBP-1 instances of size $n = 100$; instances are classified by trickiness and time distribution

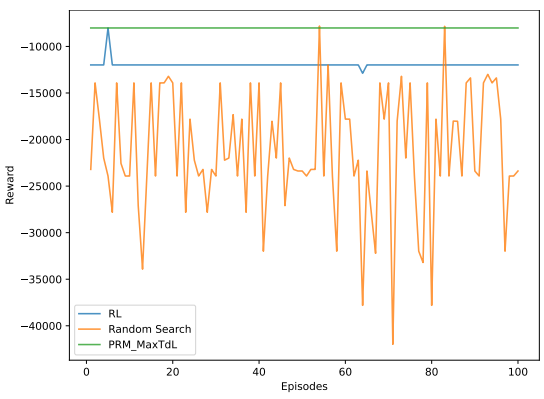
or introduce a ϵ-greedy action selection as discussed in (2.15). We argue that this must be paired with DNN hyperparameter tuning, as it can diverge from the optimal solutions indefinitely. To show this, we performed another training on the same instance as in Figures 5.3c and 5.3d with an scaled increase in network loss. The result on the same instance can be found in Figure 5.4, where we observe that the DRL agent is able to overcome local optima but gets worse solutions in later episodes. This trade-off between exploration and exploitation is frequently addressed in many heuristic design and of major interests in modern DRL study (Richard S Sutton and Andrew G Barto 2018).

In general, the sample efficiency of our DRL agent provides promising results in smaller instances, as suggested in Table 5.1. We even achieved the best overall gap of 4.65% when $n = 20$ and showed competitive results in all three trickiness categories. We also observe that optimality gaps in our DRL solutions stay in a stable range across all instance sizes, whereas PRM methods tend to achieve better gaps in larger instances, which implies that the DRL model is less subjective to instance sizes and thus has higher generalisability.

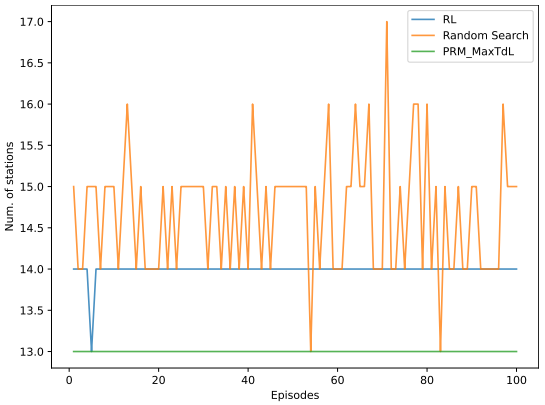
For the trickiness category, although the DRL model can still deliver competitive results in optimality gaps, we see a significant drop in performance in extremely tricky ones of size $n = 50$ and 100, suggesting its vulnerability to extended trickiness in larger instances.

Similar results can also be observed in the time distribution category. In general, all heuristics tend to have weaker performance in instances that have a time distribution with a peak in the middle. And our model is especially weak in these instances and strong in other types of time distribution. This weakness will require further investigation but we speculate that node features in the GAT model can be a significant factor.

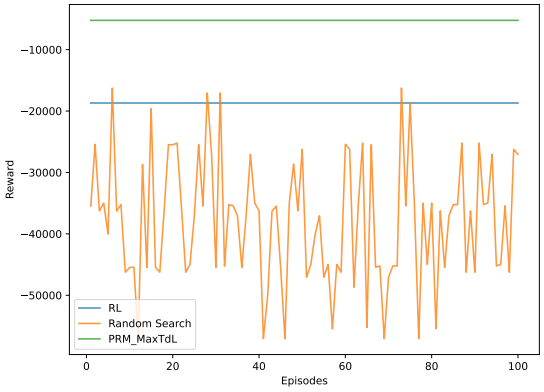
Both training and testing results of our DRL model reveal an insight towards a new solution scheme of SALBP. In general, the proposed formulation in previous sections is proved effective in small instances but requires special treatment in larger instances. A practical treatment that respects the problem-agnostic nature of model will be to add a guided initial solution search because the current scheme is highly subjective to the solution generated at the start of each episode. Thus, developing a sub-procedure that constrains the formulation of partial solutions by looking at the solution generated at the previous episode can decrease the probability of exploring bad actions.



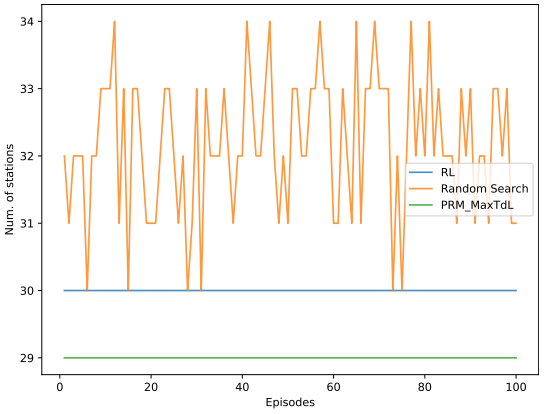
(A) Cumulative reward over 100 episodes, $n = 20$



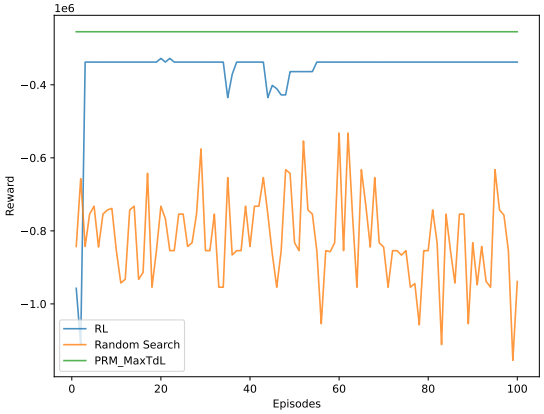
(B) Number of stations over 100 episodes, $n = 20$



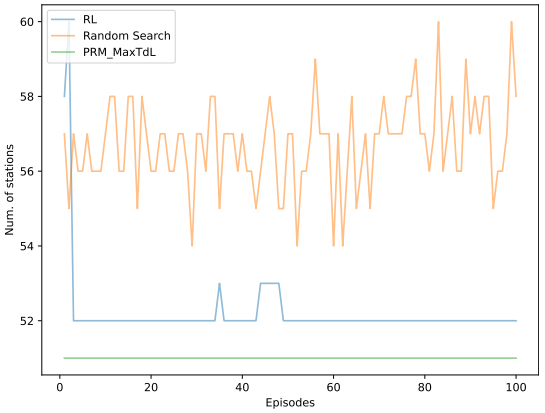
(C) Cumulative reward over 100 episodes, $n = 50$



(D) Number of stations over 100 episodes, $n = 50$

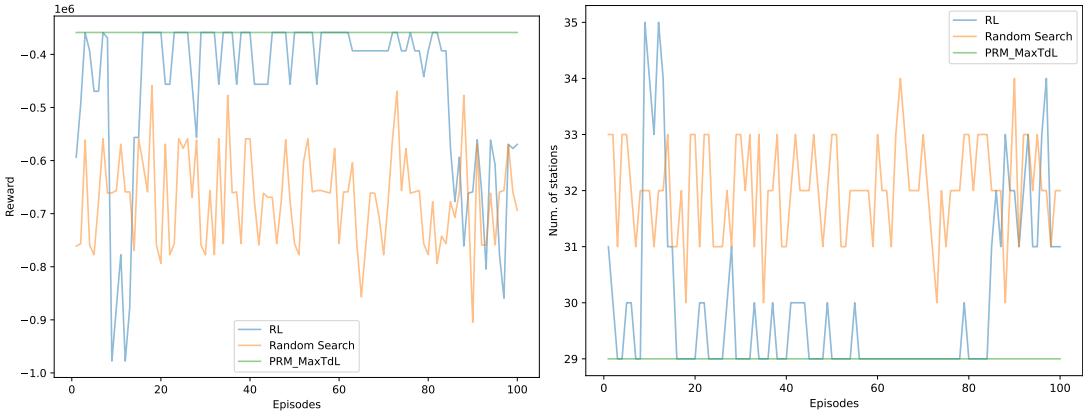


(E) Cumulative reward over 100 episodes, $n = 100$



(F) Number of stations over 100 episodes, $n = 100$

FIGURE 5.3: Training results of DRL agent on different problem sizes; green, blue and orange represent heuristic solution MaxTdL, DRL agents and randomised search, repsectively



(A) Cumulative reward over 100 episodes, $n = 50$

(B) Number of stations over 100 episodes, $n = 50$

FIGURE 5.4: Training of DRL agent with increased stochasticity on the same instance as in Figure 5.3c and 5.3d

Chapter 6

Conclusions and Future Work

This thesis investigates the practicality of incorporating DRL into constructive heuristics designs for COPs. Although the application of DRL to COPs sees a rising interest in recent literature (Mazyavkina et al. 2021), there is still no universal agreement on the context which these problems should be presented within. Given the fact there is a widespread of concepts and naming conventions in RL and COP research, the difficulty of studying one with the other is still high. Our work is therefore a combination of optimisation and RL theory under the sequential decision framework. Throughout the formalisation of this unified framework, we have shown that there are indeed many mutual connections between optimisation problems and RL. We have also distinguished the use of ML, RL, DL and DRL by examining standard formulation from their respective literature.

We have proposed a DRL model for the SALBP-1 by combining of two different DL models. The model is proven successful through a set of numerical experiments, achieving results comparable to other heuristics. We believe that our current model may be improved by incorporating a problem-specific strategy to the solution procedure. For example, one can use only the GNN to learn the precedence graph and perform heuristics to construct partial solutions without the formalisation of MDP. Alternatively, one can develop a new state representation for the instance to replace the GNN embedding and truncate network depth to achieve more stable results.

The initial results from the experiment opens a whole new area of investigation. In particular, we recognised the use of graph-based solution scheme for the SALBP-1. Although such a scheme is still at its early stage of development, we believe that it offers an alternative pathway to study ALBPs. On the other hand, the composition of action masking in our DRL model requires a deeper investigation. We believe that, when applied properly, it can serve as a constraint qualification method for solving general COPs and even a broader class of optimisation problems.

Appendix A

Algorithms

The following algorithms are adapted from Richard S Sutton and Andrew G Barto (2018):

Algorithm A.1 Policy Evaluation

Input: θ arbitrarily small
Initialise $V(s)$ and $\pi(s)$ arbitrarily
function PolicyEval(θ)
    $\Delta \leftarrow 0$
    while $\Delta < \theta$ **do**
        for $s \in S$ **do**
            $v \leftarrow V(s)$
            $V(s) \leftarrow \sum_{s' \in S} P(s' \mid s, \pi(s))[R(s, \pi(s), s') + \gamma V(s')]$
            $\Delta \leftarrow \max\{\Delta, |v - V(s)|\}$
        end for
    end while
end function

Algorithm A.2 Policy Iteration

Input: θ arbitrarily small
Initialise $\pi(s)$ arbitrarily
function PolicyIter(θ)
    policyStable $\leftarrow$ *True*
    while *policyStable* **do**
        $V \leftarrow$ PolicyEval(θ)
        for $s \in S$ **do**
            $a \leftarrow \pi(s)$
            $\pi(s) \leftarrow \arg\max_a \sum_{s' \in S} P(s' \mid s, \pi(s))[R(s, \pi(s), s') + \gamma V(s')]$
            if $a \neq \pi(s)$ **then**
                policyStable $\leftarrow$ *False*
            end if
        end for
    end while
    Return V, π
end function

Algorithm A.3 Value Iteration

Input: θ arbitrarily small
$V(s) \leftarrow 0 \, \forall s \in S$
function ValueIter(θ)
    $\Delta \leftarrow 0$
    while $\Delta < \theta$ **do**
        for $s \in S$ **do**
            $v \leftarrow V(s)$
            $V(s) \leftarrow \max_a \sum_{s' \in S} P(s' \mid s, \pi(s))[R(s, \pi(s), s') + \gamma V(s')]$
            $\Delta \leftarrow \max\{\Delta, |v - V(s)|\}$
        end for
    end while
    $\pi(s) \leftarrow \arg\max_a \sum_{s' \in S} P(s' \mid s, \pi(s))[R(s, \pi(s), s') + \gamma V(s')]$
    Return π
end function

Algorithm A.4 TD(0) tabulation

Input: π to be optimised, E batches of experience
$V(s) \leftarrow 0 \, \forall s \in S$
function TDZERO(π, E)
    for $e \in E$ **do** ▷ Roll out a batch of experience
        while $s \in e$ **do**
            $a \leftarrow \pi(s)$
            $R \leftarrow R(s, a, s')$
            $V(s) \leftarrow V(s) + \alpha[R + \gamma V(s') - V(s)]$
            $s \leftarrow s'$
            $e \leftarrow e \setminus \{s'\}$
        end while
    end for
end function

Algorithm A.5 Monte-Carlo ES

Initialise $Q(s, a)$ and $\pi(s)$ arbitrarily, $G(s, a)$ empty
function MCES(π, E)
    while termination condition not met **do**
        $E \leftarrow (s_0, \pi(s_0), s_1, \cdots, s_t)$
        for $s_i, a_i \in E$ **do**
            $G(s_i, a_i) \leftarrow \sum_{j=0}^{i-1} \gamma^j R_{j+1}$
            $Q(s_i, a_i) \leftarrow \text{AVERAGE}(G)$
        end for
        for $s_i \in E$ **do**
            $\pi(s_i) \leftarrow \arg\max_a Q(s_i, a)$
        end for
    end while
end function

Bibliography

Farley, B. and W. Clark (1954). "Simulation of self-organizing systems by digital computer". In: *Transactions of the IRE Professional Group on Information Theory* 4.4, pp. 76–84. DOI: 10.1109/TIT.1954.1057468.

Bellman, Richard (1957). "A Markovian Decision Process". In: *Journal of Mathematics and Mechanics* 6.5, pp. 679–684. ISSN: 00959057, 19435274. URL: http://www.jstor.org/stable/24900506.

Howard, Ronald A. (1960). *Dynamic Programming and Markov Processes*. 1st ed. The MIT Press.

Andreae, John (Jan. 1969). "Learning Machines: A Unified View". In: *Encyclopaedia of Linguistics, Information and Control, Pergamon Press*, pp. 261–270.

Minsky, M. and S. Papert (1969). *Perceptrons; an Introduction to Computational Geometry*. MIT Press. ISBN: 9780262630221.

Werbos, P.J. (1975). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Harvard University.

Fukushima, Kunihiko (1980). "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics* 36.4, pp. 193–202. ISSN: 1432-0770.

Carbonell, Jaime G. et al. (1983). *Machine Learning: An Artificial Intelligence Approach*. 1st ed. Symbolic Computation. Springer. ISBN: 9783662124079.

Gunther, Richard E, Gordon D Johnson, and Roger S Peterson (1983). "Currently practiced formulations for the assembly line balance problem". In: *Journal of Operations Management* 3.4, pp. 209–221. ISSN: 0272-6963. DOI: 10.1016/0272-6963(83)90005-0.

Ross, Sheldon (1983). *Introduction to Stochastic Dynamic Programming*. Probability and Mathematical Statistics: A Series of Monographs and Textbooks. Academic Press, pp. 89–106.

Sutton, Richard Stuart (1984). "Temporal Credit Assignment in Reinforcement Learning". PhD thesis.

Baybars, İlker (1986). "A Survey of Exact Algorithms for the Simple Assembly Line Balancing Problem". In: *Management Science* 32.8, pp. 909–932. DOI: 10.1287/mnsc.32.8.909.

Talbot, F., James Patterson, and William Gehrlein (Apr. 1986). "A Comparative-Evaluation of Heuristic Line Balancing Techniques". In: *Management Science* 32, pp. 430–454. DOI: 10.1287/mnsc.32.4.430.

Rumelhart, David E. and James L. McClelland (1987). "Learning Internal Representations by Error Propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*, pp. 318–362.

Barto, Andrew Gehret, Richard S Sutton, and CJCH Watkins (1989). *Learning and sequential decision making*. University of Massachusetts Amherst, MA.

LeCun, Y. et al. (1989). "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4, pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

Watkins, CJCH (1989). "Learning from delayed rewards". PhD thesis. King's College.

Zhang, Wei et al. (1990). "Parallel distributed processing model with local space-invariant interconnections and its optical architecture". In: *Appl. Opt.* 29.32, pp. 4790–4797. DOI: 10.1364/AO.29.004790.

Hochreiter, Sepp (Apr. 1991). "Untersuchungen zu dynamischen neuronalen Netzen". In.

Hornik, Kurt (1991). "Approximation capabilities of multilayer feedforward networks". In: *Neural Networks* 4.2, pp. 251–257. ISSN: 0893-6080.

Gullapalli, Vijaykumar (1992). "Reinforcement learning and its application to control". PhD thesis. University of Massachusetts at Amherst.

Sutton, Richard S (1992). "A special issue of machine learning on reinforcement learning". In: *Machine learning* 8.

Tesauro, Gerald (1992). "Practical issues in temporal difference learning". In: *Machine Learning* 8.3, pp. 257–277. ISSN: 1573-0565. DOI: 10.1007/BF00992697.

Tesauro, Gerald et al. (1995). "Temporal difference learning and TD-Gammon". In: *Communications of the ACM* 38.3, pp. 58–68. DOI: 10.1145/203330.203343.

Mitchell, Tom M. (1997). *Machine Learning*. 1st ed. McGraw-Hill Education.

Scholl, Armin and Stefan Voß (Dec. 1997). "Simple assembly line balancing—Heuristic approaches". In: *Journal of Heuristics* 2.3, pp. 217–244. ISSN: 1572-9397. DOI: 10.1007/BF00127358.

Lecun, Y. et al. (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324. DOI: 10.1109/5.726791.

Hinton, G. and T.J. Sejnowski (1999). *Unsupervised Learning: Foundations of Neural Computation*. Computational Neuroscience Series. MIT Press. ISBN: 9780262581684.

Sutton, Richard S, David McAllester, et al. (1999). "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Advances in Neural Information Processing Systems*. Vol. 12. MIT Press. DOI: 10.5555/3009657.3009806.

Bousquet, Olivier and André Elisseeff (2000). "Algorithmic Stability and Generalization Performance". In: *Advances in Neural Information Processing Systems*. Ed. by T. Leen, T. Dietterich, and V. Tresp. Vol. 13. MIT Press. DOI: 10.5555/3008751.3008778.

Kenneth P. Burnham, David R. Anderson (2002). *Model Selection and Multimodel Inference*. 2nd ed. Springer New York. ISBN: 978-0-387-95364-9. DOI: 10.1007/b97636.

Tsitsiklis, John N. (2003). "On the Convergence of Optimistic Policy Iteration". In: *J. Mach. Learn. Res.* 3.null, pp. 59–72. ISSN: 1532-4435. DOI: 10.1162/153244303768966102.

Laud, Adam Daniel (2004). "Theory and Application of Reward Shaping in Reinforcement Learning". PhD thesis. USA. DOI: 10.5555/1037402.

Hebb, D.O. (2005). *The Organization of Behavior: A Neuropsychological Theory*. Taylor & Francis. ISBN: 9781135631901.

Becker, Christian and Armin Scholl (2006). "A survey on problems and methods in generalized assembly line balancing". In: *European Journal of Operational Research* 168.3. Balancing Assembly and Transfer lines, pp. 694–715. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2004.07.023.

Hoen, Pieter Jan 't et al. (2006). "An Overview of Cooperative and Competitive Multiagent Learning". In: *Learning and Adaption in Multi-Agent Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–46. ISBN: 978-3-540-33059-2. DOI: 10.1007/11691839_1.

Bertsekas, Dimitri P. (2007). *Dynamic Programming and Optimal Control*. 3rd ed. Vol. 2. Athena Scientific. Chap. 2.

Andrés, Carlos, Cristóbal Miralles, and Rafael Pastor (2008). "Balancing and scheduling tasks in assembly lines with sequence-dependent setup times". In: *European Journal of Operational Research* 187.3, pp. 1212–1223. DOI: 10.1016/j.ejor.2006.07.044.

Hammond, David K, Pierre Vandergheynst, and Rémi Gribonval (2009). "Wavelets on Graphs via Spectral Graph Theory". In: arXiv: 0912.3848 [math.FA].

Scarselli, Franco et al. (2009). "The Graph Neural Network Model". In: *IEEE Transactions on Neural Networks* 20.1, pp. 61–80. DOI: 10.1109/TNN.2008.2005605.

Lange, Sascha, Thomas Gabel, and Martin Riedmiller (2012). "Batch Reinforcement Learning". In: *Reinforcement Learning: State-of-the-Art*. Ed. by Marco Wiering and Martijn van Otterlo. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 45–73. ISBN: 978-3-642-27645-3. DOI: 10.1007/978-3-642-27645-3_2.

Nowé, Ann, Peter Vrancx, and Yann-Michaël De Hauwere (2012). "Game theory and multi-agent reinforcement learning". In: *Reinforcement Learning*. Springer, pp. 441–470.

Bengio, Y., Aaron Courville, and Pascal Vincent (Aug. 2013). "Representation Learning: A Review and New Perspectives". In: *IEEE transactions on pattern analysis and machine intelligence* 35, pp. 1798–1828. DOI: 10.1109/TPAMI.2013.50.

Maas, Andrew L., Awni Y. Hannun, and Andrew Y. Ng (2013). "Rectifier nonlinearities improve neural network acoustic models". In: *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*.

Mnih, Volodymyr, Koray Kavukcuoglu, et al. (2013). "Playing Atari with Deep Reinforcement Learning". In: arXiv: 1312.5602 [cs.LG].

Otto, Alena, Christian Otto, and Armin Scholl (2013). "Systematic data generation and test design for solution algorithms on the example of SALBPGen for assembly line balancing". In: *European Journal of Operational Research* 228.1, pp. 33–45. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2012.12.029.

Bahdanau, Dzmitry, Kyunghyun Cho, and Y. Bengio (Sept. 2014). "Neural Machine Translation by Jointly Learning to Align and Translate". In: 1409. arXiv: 1409.0473 [cs.CL].

Grefenstette, Edward et al. (2014). "A Deep Architecture for Semantic Parsing". In: arXiv: 1404.7296 [cs.CL].

Mnih, Volodymyr, Nicolas Heess, et al. (2014). "Recurrent Models of Visual Attention". In: arXiv: 1406.6247 [cs.LG].

Otto, Alena and Christian Otto (2014). "How to design effective priority rules: Example of simple assembly line balancing". In: *Computers & Industrial Engineering* 69, pp. 43–52. ISSN: 0360-8352. DOI: 10.1016/j.cie.2013.12.013.

Esmaeilbeigi, Rasul, Bahman Naderi, and Parisa Charkhgard (Dec. 2015). "The Type E Simple Assembly Line Balancing Problem". In: *Comput. Oper. Res.* 64.C, pp. 168–177. DOI: 10.1016/j.cor.2015.05.017.

Kochenderfer, Mykel J. (2015). *Decision Making Under Uncertainty: Theory and Application*. 1st ed. MIT Lincoln Laboratory Series. The MIT Press. ISBN: 9780262029254. URL: https://mitpress.mit.edu/books/decision-making-under-uncertainty.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: *Nature* 521.7553, pp. 436–444. ISSN: 1476-4687. DOI: 10.1038/nature14539.

Schmidhuber, Jürgen (2015). "Deep learning in neural networks: An overview". In: *Neural Networks* 61, pp. 85–117. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2014.09.003.

Defferrard, Michaël, Xavier Bresson, and Pierre Vandergheynst (2016). "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering". In: arXiv: 1606.09375 [cs.LG].

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. MIT Press. URL: http://www.deeplearningbook.org.

Kipf, Thomas N. and Max Welling (2016). "Semi-Supervised Classification with Graph Convolutional Networks". In: arXiv: 1609.02907 [cs.LG].

Ruder, Sebastian (2016). "An overview of gradient descent optimization algorithms". In: arXiv: 1609.04747 [cs.LG].

Bronstein, Michael M. et al. (2017). "Geometric Deep Learning: Going beyond Euclidean data". In: *IEEE Signal Processing Magazine* 34.4, pp. 18–42. DOI: 10.1109/MSP.2017.2693418.

Gilmer, Justin et al. (2017). "Neural Message Passing for Quantum Chemistry". In: *Proceedings of the 34th International Conference on Machine Learning*. Vol. 70. Proceedings of Machine Learning Research. PMLR, pp. 1263–1272.

Hamilton, William L., Rex Ying, and Jure Leskovec (2017). "Inductive Representation Learning on Large Graphs". In: arXiv: 1706.02216 [cs.SI].

Schulman, John et al. (2017). "Proximal Policy Optimization Algorithms". In: arXiv: 1707.06347 [cs.LG].

Vaswani, Ashish et al. (2017). "Attention Is All You Need". In: arXiv: 1706.03762 [cs,CL].

Veličković, Petar et al. (2017). In: arXiv: 1710.10903 [stat.ML].

Mohri, M., A. Rostamizadeh, and A. Talwalkar (2018). *Foundations of Machine Learning*. 2nd ed. Adaptive Computation and Machine Learning series. MIT Press. ISBN: 9780262039406.

Ritt, Marcus and Alysson M. Costa (2018). "Improved integer programming models for simple assembly line balancing and related problems". In: *International Transactions in Operational Research* 25.4, pp. 1345–1359. DOI: 10.1111/itor.12206.

Sutton, Richard S and Andrew G Barto (2018). *Reinforcement Learning: An Introduction*. 2nd ed. MIT Press Academic.

Wang, Le et al. (2018). "Segment-Tube: Spatio-Temporal Action Localization in Untrimmed Videos with Per-Frame Segmentation". In: *Sensors* 18.5. ISSN: 1424-8220. DOI: 10.3390/s18051657.

Datta, Leonid (2020). "A Survey on Activation Functions and their relation with Xavier and He Normal Initialization". In: arXiv: 2004.06632 [cs.LG].

Dwivedi, Vijay Prakash et al. (2020). "Benchmarking Graph Neural Networks". In: arXiv: 2003.00982 [cs.LG].

Huang, Shengyi and Santiago Ontañón (2020). "A Closer Look at Invalid Action Masking in Policy Gradient Algorithms". In: arXiv: 2006.14171 [cs.LG].

Hubbs, Christian D et al. (2020). "Or-gym: A reinforcement learning library for operations research problems". In: arXiv: 2008.06319 [cs.AI].

Moerland, Thomas M., Joost Broekens, and Catholijn M. Jonker (2020). "Model-based Reinforcement Learning: A Survey". In: arXiv: 2006.16712 [cs.LG].

Kirk, Robert et al. (2021). "A Survey of Generalisation in Deep Reinforcement Learning". In: arXiv: 2111.09794 [cs.LG].

Mazyavkina, Nina et al. (2021). "Reinforcement learning for combinatorial optimization: A survey". In: *Computers & Operations Research* 134, p. 105400. DOI: 10.1016/j.cor.2021.105400.