Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

State-space decomposition for Reinforcement Learning

Author:
Esther Wong

Supervisor:
Dr. Kin Leung

Second Marker:
Dr. Tony Field

August 2, 2021

Abstract

To this day, Deep Reinforcement Learning (DRL) has shown promising results in research and is gradually emerging into many real-life applications. However, the applicability of DRL is often limited by its excessive training time as many problem settings suffer from the state-space explosion problem. In this research, we develop State-Space Decomposition Reinforcement Learning (SSD-RL), a novel DRL method which helps alleviate this issue by splitting up large state spaces into smaller ones through state-space decomposition, thereby allowing us to distribute computation and accelerate training. The key idea is to use smaller neural networks to learn the dynamics of the decomposed state sub-spaces, whilst another neural network considers the relatively less frequent interactions between the different state sub-spaces.

We applied SSD-RL to a variety of environments, including one based off Alibaba's cluster dataset where the agent learns to how to efficiently allocate processing power in data centers. Through our experiments, we successfully show that SSD-RL notably reduces training time for problems with an appropriate decomposability factor compared to other state-of-the-art methods. We also investigate the condition and parameter settings under which the SSD-RL can be most beneficial in terms of reducing learning time. Finally, we demonstrate how SSD-RL can be extended into a distributed method, allowing us to overcome some of the hurdles of running DRL in distributed systems. From this study, we conclude that applying state-space decomposition to Reinforcement Learning problems is indeed effective for certain problem settings, and the idea of decomposing the learning into multiple neural networks brings us one step closer to being able to create distributed Reinforcement Learning techniques.

Acknowledgements

I would like to express my sincere gratitude to Dr. Kin Leung for inspiring me with his passion for research and providing invaluable guidance throughout the whole project. You have made this project a very rewarding and enjoyable experience.

I also want to thank my parents for their unconditional love and encouragement. You have both supported me immensely throughout my entire degree.

Lastly, I am grateful for all my friends who have walked this challenging journey with me. Your support and banter have kept me going.

Contents

1 Introduction **5**
- 1.1 Contributions . . . 5
- 1.2 Outline . . . 6

2 Technical background **7**
- 2.1 Machine Learning (ML) . . . 7
- 2.2 Artificial Neural Networks (ANN) . . . 7
  - 2.2.1 Hyper-parameter tuning . . . 8
- 2.3 Reinforcement Learning (RL) . . . 9
  - 2.3.1 Markov Decision Processes (MDP) . . . 9
  - 2.3.2 Optimisation problem . . . 9
  - 2.3.3 Reinforcement Learning algorithms . . . 10
  - 2.3.4 Deep Reinforcement Learning (DRL) . . . 11
  - 2.3.5 State-space explosion problem . . . 13

3 Related work **14**
- 3.1 Hierarchical Reinforcement learning (HRL) . . . 14
  - 3.1.1 Methods to formulate problem hierarchy . . . 15
- 3.2 Multi-agent Reinforcement learning . . . 15
- 3.3 Simplifying the state-space . . . 16
- 3.4 Summary . . . 17

4 A new approach: State-space decomposition in Reinforcement Learning **19**
- 4.1 Challenges . . . 20

5 Developing SSD-RL using grid-world environments **21**
- 5.1 Grid-world environments . . . 21
- 5.2 Initial experiments . . . 22
- 5.3 Performance evaluation . . . 24
  - 5.3.1 Reward curves . . . 24
  - 5.3.2 Effects of changing the state-space size . . . 25

6 SSD-RL: State-space Decomposition Reinforcement Learning **26**
- 6.1 Network architecture . . . 26
- 6.2 Training . . . 27
  - 6.2.1 Stage 1: Training within state sub-spaces . . . 27
  - 6.2.2 Stage 2: Training across state sub-spaces . . . 27
- 6.3 Decomposed replay buffer . . . 28

7 Decomposability factor for SSD-RL **30**
- 7.1 Definition . . . 30
- 7.2 'Two goals' environment . . . 30
- 7.3 Effects on performance gain . . . 31

8 Applying SSD-RL to Alibaba's dataset **33**
8.1 Data processing . . . 33
8.2 Creating the 'Workload Distribution' environment . . . 33
8.3 Evaluation of results . . . 35
8.4 Effect of varying the decomposability factor . . . 35

9 Applicability of SSD-RL **37**
9.1 Suitable environments . . . 37
9.1.1 Feasibility of state space decomposition . . . 37
9.1.2 Decomposability factor . . . 37
9.1.3 Size of state-space . . . 37
9.1.4 Reward function . . . 37
9.2 Hyper-parameters to tune . . . 38
9.3 Ease of use . . . 39
9.4 Summary . . . 39

10 Extending SSD-RL **40**
10.1 Distributed SSD-RL . . . 40
10.2 Multi-agent SSD-RL . . . 41
10.3 Future work . . . 41

11 Conclusion **43**
11.1 Summary . . . 43
11.2 Ethical discussion . . . 43

A Hyper-parameters used in experiments **45**
A.1 Baseline method: Deep Q-Learning . . . 45
A.2 Our new method: State-space decomposition reinforcement learning . . . 45

List of Figures

2.1 Structure of an artificial neuron . . . 8
2.2 Artificial Neural Network architecture with hidden layer . . . 8
2.3 Reinforcement Learning loop . . . 9
2.4 Deep Reinforcement Learning architecture showing the inputs being the state, and the output being the state-action values for each action in the input state. . . . 12

3.1 Example of sub-goal formulation for the *rooms example* in Sutton et al.'s work. G_1 and G_2 indicate the sub-goals. [22] . . . 14
3.2 CMARL: Neural network architecture for medical image landmark detection in Leroy et al.'s work [27]. . . . 16
3.3 Reinforcement Learning with PCA compressed state-space from the work of Curran et al. [31] . . . 17

4.1 Decomposing state space $\mathcal{S}$ into sub-spaces $S_1, ..., S_4$ through simplifying the transition probability matrix $\mathcal{P}$ . . . 19
4.2 Network architecture of our new approach . . . 20

5.1 Room maze environment . . . 21
5.2 Comparison of loss curves between SSD-RL with stratified learning and SSD-RL with unified learning. . . . 22
5.3 ϵ value during training with Double-start linear decay . . . 24
5.4 Comparison of return curves between SSD-RL and Deep Q-learning for the two variations of the room maze environment. Results are averaged over 3 runs. . . . 24
5.5 Performance of SSD-RL and DQL for different grid sizes in the sub-goal room maze environment . . . 25

6.1 SSD-RL Network architecture. The + connections represent skip connections. . . . 26
6.2 Sampling method of the decomposed replay buffer for batch size $B + B_C$ . . . 28

7.1 Two goals environment. The dotted line indicates the split between the first state sub-space and the second one. . . . 30
7.2 Performance analysis for different ϵ values in the Two goals environment. Each point is averaged over 5 runs. . . . 31

8.1 Workload Distribution environment . . . 34
8.2 Performance comparison between SSD-RL and DQL with the Workload Distribution environment with $\epsilon = 0.2$ . . . 35
8.3 Effect of ϵ on performance of SSD-RL. Results are averaged over 5 runs. . . . 36
8.4 Comparison of convergence time between SSD-RL and DRL for different values of ϵ in the 'Workload Distribution' environment. . . . 36

9.1 Effect of changing N on SSD-RL's performance. We ran this experiment on the Two goals environment with varying ϵ values. . . . 38
9.2 Difference between a tuned T value ($T = 15$) and a non-tuned T value ($T = 0$) . 39

10.1 Illustration of SSD-RL as a distributed Reinforcement Learning algorithm . . . 40
10.2 Exploration visualisation for Multi-agent SSD-RL. S_k denotes the kth state sub-space 41

Chapter 1

Introduction

Research in Deep Reinforcement Learning (DRL) has been prominent over the past few years, allowing us to solve complex problems in the realms of robotics, self-driving cars [1] and resource management [2]. DRL is a type of machine learning where an agent explores an environment consisting of different states, while learning from reward signals emitted from each interaction. As the agent explores, it learns what actions are good to take in its current state, and gains a better understanding of how to maximise its long term gains. Essentially, the agent learns from experience, and does not require a labelled data set like other types of machine learning techniques. However, with big complex problems, our state and action spaces become extremely large, leading to the state-space explosion problem. This causes exponentially long training times and limits the practicality of using DRL for such problems in real-life [3].

For example, there has been past research on using DRL to control Software Defined Network (SDN) controllers [4]. The goal of SDN controllers is to strategically control network traffic in the most efficient way. In a world that's becoming more connected each day, networks are constantly expanding, and we end up dealing with enormous state spaces as there are endless configurations our network can be in at any point in time. The lengthy training times cause the controller to predominantly make sub-optimal decisions, which defeats the purpose of using DRL. Another major problem is that the data we use to learn is stored in a distributed manner. Due to the size of SDNs, the network's nodes are often stored across multiple machines. With the lack of distributed DRL algorithms, we encounter two issues: firstly, there is a heavy communication cost to transfer all the data to a central entity to carry out DRL, and secondly, there may be privacy constraints during transmission. Therefore, the real-life applicability of DRL to this problem is limited.

In our research, we propose a novel DRL approach that addresses two problems: the state-space explosion issue which causes excessive training times, and the limitation of using DRL due to large amounts of data being stored in a distributed manner. Our key idea is to split up a large Reinforcement Learning problem into several smaller ones by decomposing the state-space of our environment. We can decompose the state-space because the scope of interactions within large environments is often sparse, which allows us to define disjoint regions where most of the important interactions occur. Through this, we are able to split up training into multiple smaller neural networks, allowing us to parallelise our computation in distributed environments and significantly speed up training. Taking the SDN controller example again, each node in the network may be part of a specific sub-network, meaning that most of the interactions occur within each sub-network instead of across different sub-networks. In this case, the question would be if we could train on each sub-network separately, and combine these learned sub-problems to solve the over-arching problem.

1.1 Contributions

- **State-space decomposition Reinforcement Learning (SSD-RL):** We develop a novel DRL method that utilises state-space decomposition to speed up training for problems with an appropriate decomposability factor. Through multiple environments, we demonstrate that SSD-RL converged up to seven times faster than other state-of-the-art methods.

- **Distributed SSD-RL:** We propose a method that extends SSD-RL into a distributed Rein-

forcement Learning algorithm. This extended method tackles the problem that many real-life environments are hosted in distributed systems, which constrains the effectiveness of DRL due to the lack of data locality.

- **Decomposability factor:** We define a quantifiable technique to evaluate how well an environment is suited for SSD-RL. We also obtain a decomposability factor threshold for several environments in which SSD-RL outperforms current state-of-the-art methods.
- **Apply SSD-RL to several datasets:** We apply our developed technique to multiple environments including grid-world environments and one created using Alibaba's cluster trace data set. In the latter environment, our agent learns how to allocate incoming workload to different data centers in the most optimal way. SSD-RL is shown to be more efficient and stable than other state-of-the-art methods, and has proven to be able to help us tackle large real-life problems.

1.2 Outline

Firstly, we provide preliminaries for DRL in chapter 2. Then, we discuss related work and the idea of using state-space decomposition for DRL in chapters 3 and 4. Next, we document the process of developing SSD-RL through experimenting with several simple environments in chapter 5. In chapter 6, we provide a detailed technical description of how our final version of SSD-RL is implemented. We then evaluate SSD-RL against a state-of-the-art method using a real-life dataset in chapter 8. Chapters 7 and 9 discusses the practicalities of using SSD-RL for more efficient learning. Finally, chapter 10 presents how SSD-RL can be extended into a distributed Reinforcement Learning method, and also discusses any future work that can be done to develop our method even further.

Chapter 2

Technical background

In this chapter, we look at the fundamental technical concepts that are needed in this project.

2.1 Machine Learning (ML)

Machine Learning (ML) is a method which allows systems to learn and improve from experience. It aims to model something with data and use this knowledge to make decisions. It is currently a big area of research today, and it is also widely used in commercial products. For example, many shopping websites utilise your past preferences to intelligently predict and recommend new products that may interest you [5].

There are three main types of Machine Learning [6]:

- **Supervised learning:** When a computer learns from correctly labeled data. Typically, the program will approximate a function to describe the input data, allowing it to make good predictions for unseen data. The two main types of supervised learning problems are:
  - **Linear regression**: We aim to predict a numerical label for a given test sample.
  - **Classification**: We aim to predict a class label for a given test sample.
- **Unsupervised learning:** When a computer learns from data with no labels. The program tries to learn hidden patterns in the data through various algorithms such as clustering [7].
- **Reinforcement learning:** The computer learns from interacting with an environment which gives it reward signals for each action it takes. Possible reward signals could be a 'good' or 'bad' label, or a numerical value.

Supervised learning and unsupervised learning have been developed extensively over the past years, and is now being used effectively in many everyday applications. Reinforcement learning, on the other hand, has been a popular topic of research lately, and has only just begun to emerge in real-life applications [8]. Therefore, in this project, we chose to focus on speeding up Reinforcement learning so that it can be used more successfully in the real world.

2.2 Artificial Neural Networks (ANN)

Many Machine Learning techniques, including ones in reinforcement learning, utilize artificial neural networks. Artificial neural networks are constructed with artificial neurons, and they are used to approximate a function f which describes the relationship between our input data and what we want to predict. Each individual neuron is structured as shown in figure 2.1. The output of the neuron can be calculated by equation 2.1 [9].

$$y = g(\theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + b) \tag{2.1}$$

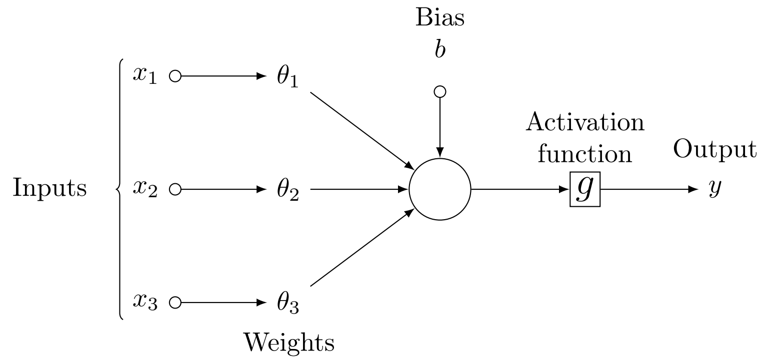

Figure 2.1: Structure of an artificial neuron

In reality, we utilise neural networks to approximate much more complex problems, and we usually require a larger set of parameters θ to come to a relatively accurate prediction. Therefore, it is common to use several layers of neurons to form the neural network architecture. See Figure 2.2.

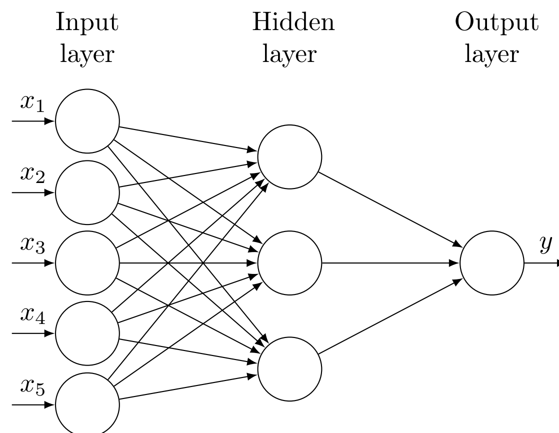

Figure 2.2: Artificial Neural Network architecture with hidden layer

In order to update the parameters to approximate our function f, we commonly use a method called stochastic gradient descent - an iterative method to optimise an objective function [10]. Firstly, we pass a training sample x_t through the neural network using forward propagation, obtaining an output value $h(x_t) = y_t$. We then compare our prediction y_t to the correct value y, and describe the correctness of our prediction using a cost function such as Mean Squared Error (MSE). We then calculate the gradients of the cost function with respect to our current parameters θ. This gradient is back-propagated through the neural network, which adjusts the parameters, and gradually helps the neural network converge to a good approximation of f, with $h \approx f$.

2.2.1 Hyper-parameter tuning

In reality, a lot of other things need to be considered when utilizing an ANN. There are many hyper-parameters which need to be tuned for a specific problem [10]. For example, each time we perform gradient descent, we need to specify a learning rate which indicates how much we want to adjust our parameters based on our current training sample. A learning rate that is too large may cause our approximation to never converge, a learning rate too small may cause our solution to end up in a local minimum rather than the global minimum. Some other parameters we need to consider include activation functions, number of hidden layers, and number of epochs.

2.3 Reinforcement Learning (RL)

Reinforcement Learning (RL) is a type of machine learning that has had many recent breakthroughs. It has proven to be able to learn difficult tasks in the realms of robot control, traffic management, gaming and more [1, 11]. RL essentially helps an agent learn how to maximize its rewards in a given environment. The agent learns through interacting with the environment and observing the different rewards or penalties it receives. With sufficient training, agents are able to learn how to navigate the environment in order to maximise rewards.

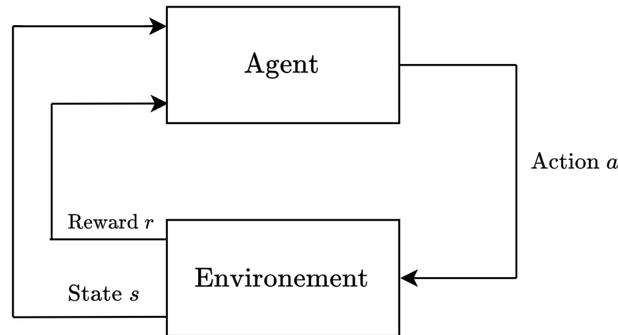

Figure 2.3: Reinforcement Learning loop

2.3.1 Markov Decision Processes (MDP)

Each RL problem assumes that there is an underlying Markov Decision Process (MDP). MDPs are used to mathematically model decision making control processes, and forms the basis of how we derive the optimization problem for RL. All MDPs satisfy the Markov property, which indicates that any transition to another state is only dependent on the current state, irrespective of the previous states [12].

MDPs are defined by a quintuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ [13]:

- **State space** ($\mathcal{S}$): A state is a representation of the environment. The state space is the set of all possible states.
- **Action space** ($\mathcal{A}$): The set of all possible actions the learning agent can take in the environment.
- **Probability transition matrix** ($\mathcal{P}$): Each element in this matrix can be defined as $\mathcal{P}^a_{ss'}$, which is the probability of transitioning from a state s to another state s' when taking an action a. This matrix has dimensions $|\mathcal{S}| \times |\mathcal{S}| \times |\mathcal{A}|$.
- **Reward matrix** ($\mathcal{R}$): Each element in this matrix can be defined as $\mathcal{R}^a_{ss'}$, which is the reward received when transitioning from state s to state s' by taking action a. This matrix has dimensions $|\mathcal{S}| \times |\mathcal{S}| \times |\mathcal{A}|$.
- **Reward discount factor** (γ): When making decisions, we want to account for future rewards we can gain. This allows us to take less intuitive actions early on which lead to much larger rewards in the future. γ defines how much we care about future rewards relative to immediate rewards. γ is a real value between 0 and 1. If $\gamma = 1$, we care about all future rewards. On the other hand, having $\gamma = 0$ causes us to completely neglect future rewards, and focus on making the best action at this current moment in time.

2.3.2 Optimisation problem

In Reinforcement Learning, we want to learn an optimal policy (a strategy for choosing actions) that enables an agent to maximise its expected total rewards when interacting with the environment.

A policy π is defined as a mapping between a state s and the probability distribution over each possible action in $\mathcal{A}$. The larger the probability of the action, the more likely the agent will choose to take that action in state s. We can formally define the agent's expected total reward as the return R_T in equation 2.2, where γ is the discount factor, r is the instantaneous reward received at time step t, and T is the total time steps in an episode of training [13].

$$R_T = \sum_{t=0}^{T} \gamma^t r_t \tag{2.2}$$

Value function

To solve for the optimal policy, we estimate a value function, which indicates how good it is to be in a current state for a given policy π. We use the expected returns as a metric to calculate each state's value [13].

$$\begin{aligned} V^\pi(s) &= \mathbb{E}[R_t | S_t = s] \\ &= \sum_{a \in \mathcal{A}} \pi(s|a) \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} (\mathcal{R}^a_{ss'} + \gamma V^\pi(s')) \end{aligned} \tag{2.3}$$

A policy π is defined to be strictly better than another policy π' if $V^\pi(s) > V^{\pi'}(s)$ [13]. For finite MDP problems, there is always one optimal policy, and it can be represented by the optimal value function defined in equation 2.4.

$$\begin{aligned} V^*(s) &= \max_\pi V^\pi(s) \\ &= \max_a \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} (\mathcal{R}^a_{ss'} + \gamma V^*(s')) \end{aligned} \tag{2.4}$$

State-action function (Q-value function)

Optimal value functions also share the same state-action value function (2.6). The state-action value gives us the expected returns from taking action a in state s (2.5) [13]. They are also often referred to as Q-value functions.

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}[R_t | S_t = s, A_t = a] \\ &= \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} (\mathcal{R}^a_{ss'} + \gamma \sum_{a'} \pi(s'|a') Q^\pi(s', a')) \end{aligned} \tag{2.5}$$

$$\begin{aligned} Q^*(s,a) &= \max_\pi Q^\pi(s,a) \\ &= \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} (\mathcal{R}^a_{ss'} + \gamma \max_{a'} \pi(s'|a') Q^*(s', a')) \end{aligned} \tag{2.6}$$

Optimal policy

In order to extract the optimal policy π^* from the optimal state-action function Q^*, we compute equation 2.7 [14].

$$\pi^* = \arg\max_{a \in \mathcal{A}} Q^*(s,a), \quad \forall s \in \mathcal{S} \tag{2.7}$$

2.3.3 Reinforcement Learning algorithms

In order to estimate the value function and derive an optimal policy for our problem, we can use various iterative methods such as Value iteration or Policy iteration [15]. Both of these solve the reinforcement learning problem by calculating the value function iteratively according their equations in (2.3). However, these methods require us to have access to the full probability transition matrix $\mathcal{P}$ and reward matrix $\mathcal{R}$ of the MDP, which is often not obtainable in real-life. This lead to the development of other methods which handle this scenario. We can generally split these methods into two categories [16]:

1. **Model-based learning**: We learn the underlying model by calculating estimates of $\mathcal{P}$ and $\mathcal{R}$. Once we have a good model, we use this to plan our actions and find a good policy.

2. **Model-free learning**: We don't attempt to learn the underlying model ($\mathcal{P}$ or $\mathcal{R}$). Rather, we directly use experienced samples to estimate the optimal state-action values to form an optimal policy.

The advantages to model-based learning is that it is more data efficient. Models are also transferable to similar tasks, meaning that they can learn similar tasks very efficiently because we already have an underlying model to work with. However, with model-based learning, the agent can only get as good as the model we have learned. We are limited by the accuracy of the model, which can lead to compounding errors. With model-free learning, we don't have this problem. The state-action values we learn are fit directly to observed data, so we are only limited by the data. But because of this, an abundance of data is needed in order for model-free methods to learn well.

Computationally, model-free learning methods are more efficient. Since we are estimating the state-value function, our memory cost is $|\mathcal{S}| \times |\mathcal{A}|$. With model-based learning, we attempt to learn the underlying model which has a larger memory cost of $|\mathcal{S}| \times |\mathcal{S}| \times |\mathcal{A}|$.

Exploration vs. exploitation

A common challenge in Reinforcement Learning algorithms is to ensure that the agents explore the environment enough. Insufficient exploration will lead to convergence to a local minimum solution, rather than the global minimum solution. However, if our agent has explored enough and has a clear understanding of its current area in the environment, we might want to make it exploit what is has learned in order to progress and move on to try and find more rewards. Therefore, there is a trade-off between exploration and exploitation. A well-known method to address this issue is to make the agent follow an ϵ-greedy policy whilst training (shown in equation 2.8) [17].

$$\pi(s|a) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|}, & \text{if } a \text{ is the optimal action} \\ \frac{\epsilon}{|\mathcal{A}|}, & \text{otherwise} \end{cases} \tag{2.8}$$

ϵ is a hyper-parameter we can tune for training. When ϵ is large, we explore more as there is a larger probability that we won't take what we think is the optimal action. When ϵ is small, we exploit our knowledge more. In many algorithms, ϵ starts off large and decays after each episode to help learning converge [17, 18]. This way, we can balance exploration and exploitation and make sure we are never ruling out one or the other.

2.3.4 Deep Reinforcement Learning (DRL)

Many Reinforcement Learning problems may have a continuous state space - for example, the GPS co-ordinates of a car. In this case, it would be infeasible to use tabular Reinforcement Learning methods to solve the problem as there would be an infinite number of states. Deep Reinforcement Learning (DRL) was introduced to solve this issue. DRL is a technique which combines Reinforcement Learning and Deep Learning. It utilises artificial neural networks to approximate the state-action value function (See figure 2.4). DRL has shown to be successful in many problems. It has been used to solve problems such as Atari games, and has even managed to surpass a human expert on certain games [11]. There have also been publications on the effectiveness of DRL on robot control, showing that certain locomotion tasks can be learned within five minutes of training [19].

Value function approximation methods

One of the most widely used value function approximation methods in DRL is Deep Q-learning, shown in algorithm 1 [11]. It was originally proposed by Minh et al. and is a variant of the Q-learning algorithm previously introduced by Sutton et al. [13] Deep Q-learning is a model-free method as it solves the Reinforcement Learning problem through direct sampling, without estimating any parts of the underlying MDP model. As the agent explores the environment, it updates a Q-network Q_θ which is an artificial neural network that approximates the state-action value function. After adequate training, we converge to the optimal value, giving us $Q_\theta(s, a) \approx$

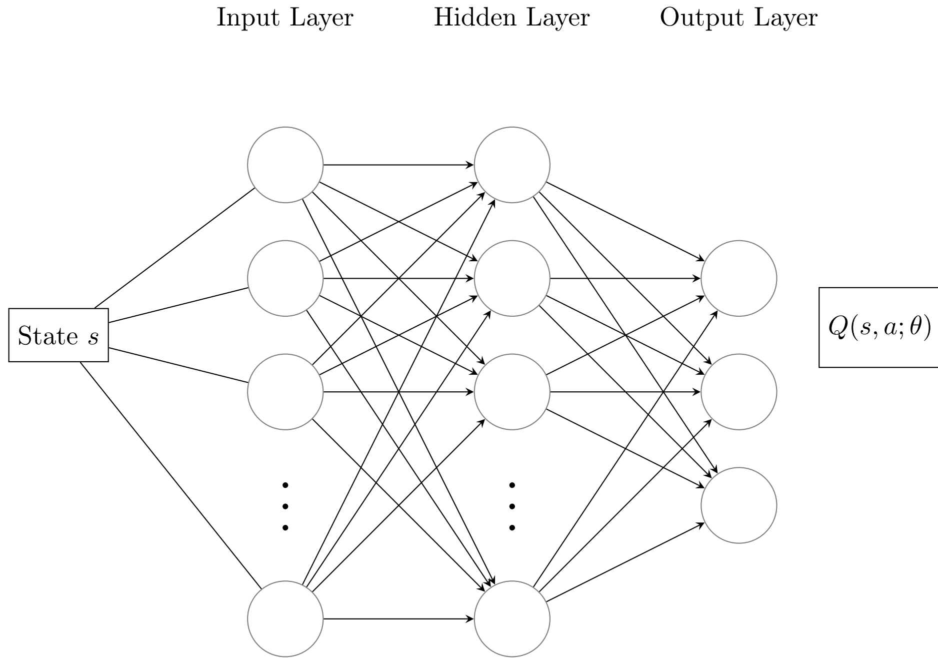

Figure 2.4: Deep Reinforcement Learning architecture showing the inputs being the state, and the output being the state-action values for each action in the input state.

$Q^*(s,a)$. Similar to basic Reinforcement Learning, we can derive the optimal policy by taking the maximum state-action value for each state.

Deep Q-learning achieves excellent stability by introducing an *experience replay buffer*. This is used to store the agent's experience at each time-step during training. Experiences are often referred to as transitions, which are represented by a tuple (s_t, a_t, r_t, s_{t+1}). In the main loop of the algorithm, the agent randomly sample batches from this buffer to learn from [11]. This feature stabilises the learning as we update the Q-network based on more samples and prevents previously learned knowledge from being overwritten.

Algorithm 1: Deep Q-learning

```
Initialise a Q-network Q_θ with random weights θ;
for each episode do
    S ← S_init;
    for each step in episode do
        Choose action A from current state using ε-greedy policy from Q_θ;
        Take A and observe reward R and next state S';
        Store transition experience (S, A, R, S') in experience replay buffer;
        Sample mini-batch B of size N;
        θ ← θ − α (1/N) Σ_{b∈B} ∇_θ[r + γ max_a Q_θ(S', A) − Q_θ(S, A)]^2;
        S ← S';
    end
end
                                        ▷ ∇_θ denotes the gradient w.r.t. θ
```

Policy gradient methods

In problems where there is a continuous action space, it is sometimes easier to utilise policy gradient methods. Instead of approximating the state-action function, we can directly approximate the policy using the neural network [20]. These methods are usually more straight-forward and contain significantly fewer parameters compared to value approximation methods. They are also proven to be able to converge to a local optimum policy [20]. However, with problems that

require significant exploration or complex policies, the globally optimum policy is rarely reached. Therefore, it is usually better to use value function approximation methods in these cases.

2.3.5 State-space explosion problem

A recurring challenge in Reinforcement Learning has been solving problems with a very large state or action space. Given a state and action space, there are at most $|\mathcal{A}|^{|\mathcal{S}|}$ unique policies. This means that the size of the problem's solution space grows exponentially with each additional feature in our state [3]. This is commonly described as the "curse of dimensionality". With complex problems, we can clearly see that training time can become very unrealistic.

In this project, we aim to find and evaluate a general technique which addresses this problem in Deep Reinforcement Learning.

Chapter 3

Related work

In this chapter, we discuss existing research that tackle the state- space explosion problem discussed in 2.3.5. We examine their advantages and disadvantages and discuss how they have informed our approach.

3.1 Hierarchical Reinforcement learning (HRL)

Hierarchical Reinforcement Learning (HRL) is a category of Reinforcement Learning methods which aim to decompose complex tasks into simpler sub-tasks. Not only do they simplify a problem, but they also allow for quick adaptation to new problems if the sub-tasks are general and reusable [21].

Several works have proven that training multiple layers of policies has been able to learn difficult tasks more efficiently [22, 21, 23]. With these methods, there are usually lower-level policies which directly apply actions to the environment, and there are higher-level policies which use the lower-level policies to perform long-term planning. An early but popular HRL framework was proposed by Sutton et al. in [22]. They defined a concept called *options*, which is a course of actions consisting of a policy, a termination condition and an initiation set. Examples of options include opening the door, picking up an object, eating lunch etc. This framework allowed the formulation of sub-goals, and showed that they could be used with common Reinforcement Learning algorithms like Q-learning [22]. Figure 3.1 shows an example of sub-goal formulation in Sutton at el.'s work. They showed that through using options, the agent was able to plan its actions at a room-by-room level, rather than a step-by-step level when using primitive actions only, making it much faster.

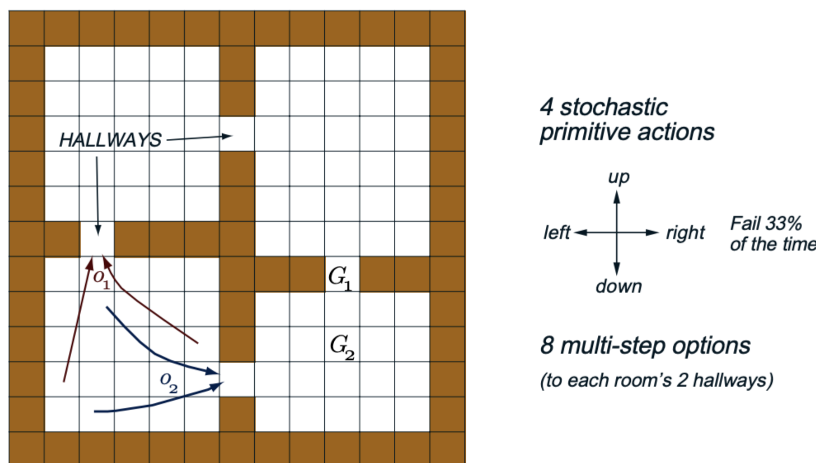

Figure 3.1: Example of sub-goal formulation for the *rooms example* in Sutton et al.'s work. G_1 and G_2 indicate the sub-goals. [22]

3.1.1 Methods to formulate problem hierarchy

With many of the current HRL methods, we require manual sub-task formulation specific to each problem. However, this can be difficult to apply to real-life problems which we don't have a lot of prior knowledge about. This issue has been addressed by several works.

Learning the problem hierarchy

Nachum et al. proposed a method in which we can generalise sub-task proposing methods [24]. Instead of setting a tailored sub-goal for lower-level policies, they propose an architecture (HIRO) in which lower-level controllers are supervised with goals that are learned and proposed automatically by the higher-level controllers. They evaluated HIRO on various difficult environments relating to navigation and found that the majority of agents were able to converge to a relatively good solution within only a few million experience samples, outperforming other non-HRL and HRL methods [24]. Nachum et al. has taken us a step closer to generalising HRL methods. However, work has still yet to be done to make their method more stable and applicable to more problems.

Identifying irrelevant states

The work of Jong et al. proposes a method in which we try to identify irrelevant states in a previously solved problem, so that we can define the hierarchy of a similar problem much more easily [25]. Instead of using traditional Reinforcement Learning methods, Jong et al. chooses to use a form of Bayesian Reinforcement Learning where the agent learns a distribution of possible values for the MDP problem (transition probability matrix). After a sufficient amount of training, they sampled 100 MDPs from the learned distribution and observed for irrelevant states in the optimal policies. After finding state abstractions, they applied them to similar problems and found that learning was more stable and faster. This is expected since the state abstractions used in their experiments successfully reduced the size of the state space from 500 to 300 [25]. The main cost of the method is that searching for state abstractions takes a lot of computational time. It is most beneficial to use this method when the cost of taking an action is relatively high and the cost of computation is relatively low. However, if state abstraction is successful, it significantly simplifies similar problems and allows us to learn them quickly.

3.2 Multi-agent Reinforcement learning

Currently, we have discussed methods in which one agent solves a problem through Reinforcement Learning. However, several works [26, 27] have explored the possibility of having several agents which cooperate in order to learn the solution more efficiently in complex problems. Multi-agent Reinforcement Learning is a category of Reinforcement Learning methods which utilise more than one agent to learn.

Methods in tabular Reinforcement Learning

Tan successfully found three main ways in which multiple agents can cooperate in a tabular environment: by communicating instantaneous information, episodic experience, and learned knowledge [26]. It was found that through cooperating, the agents were able to converge to a solution faster than individual agents as they were able to explore different parts of the state space, and communicate any knowledge learned to the other agents. One of the considerations of using this method is that the extra information received from other agents actually enlarges the state space. Hence, with certain problems, it can often be detrimental to share large amounts of knowledge with each other. However, by selecting important information to share, we can tackle complex problems with large state space in a more efficient manner.

Methods in Deep Reinforcement Learning

In the area of Deep Reinforcement Learning, Leroy et al. proposed a multi-agent method (CMARL) which was shown to be effective in medical image landmark detection [27]. Each agent explores a 3D medical scan and aims to find important landmarks. During this process, the agents cooperate using both implicit and explicit communication methods.

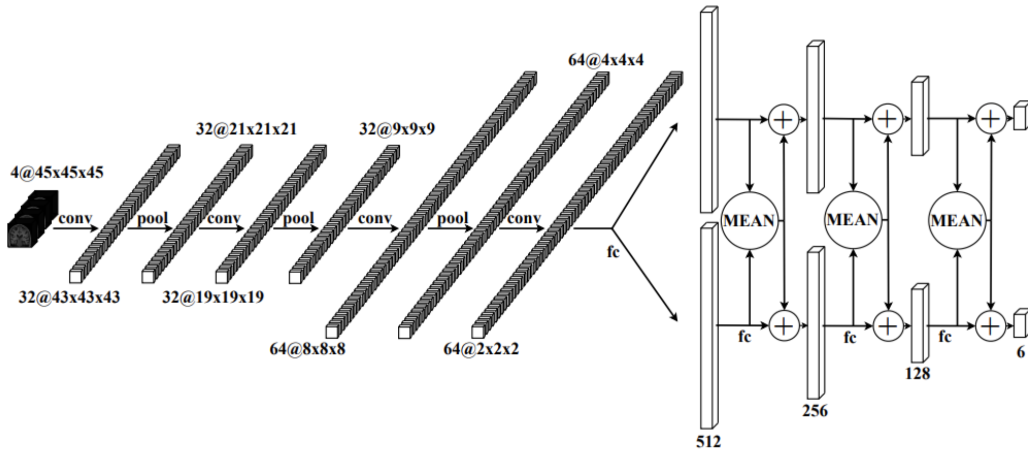

Figure 3.2: CMARL: Neural network architecture for medical image landmark detection in Leroy et al.'s work [27].

- **Implicit communication**: The agents share communication channels in CMARL's convolutional layers. This means that every learning update made by an individual agent can be propagated to other agents as well. This is shown in figure 3.2.
- **Explicit communication**: CMARL takes the average of each output of the fully connected layers for each agent, and feeds those values into the input of the next fully connected layer. This is also shown in figure 3.2.

The CMARL architecture was shown to outperform single agents as it was able to detect landmarks in medical images more accurately [27].

Evaluation

Multi-agent methods exploit parallised learning, which allows for faster convergence and even higher accuracy as seen with CMARL [27]. However, Multi-agent learning can be practically limited for fields which require expensive physical hardware. For example, it can be unrealistic to purchase several highly accurate robots in order to carry out Multi-agent Reinforcement Learning. There is also a communication cost associated in many of the methods. When we exchange information between agents too frequently, it could worsen training time [26]. Therefore, there are various parameters we need to tune and most importantly, we must consider if the nature of our problem is practically suitable for Multi-agent methods.

3.3 Simplifying the state-space

State abstraction

State abstraction is a process that maps an original state representation into one that is more compact and easier to work with. It aims to distinguish between relevant and irrelevant information in a problem. Through abstracting, learning large real-life problems become more feasible. One of the earliest works on state abstractions by Dean et al. introduces a method to partition a MDP into a Bounded MDP, which consists of several abstracted MDPs which behave approximately the same under a set of policies [28]. This partitioning condition is known as bisimulation. Bounded MDPs capture which MDPs our agent *could* be in, given its current knowledge. They also defined an algorithm called Interval Value Iteration (IVI), which given a Bounded MDP, computes the bounds of the optimal value function. With this, we can extract a *pessimistic optimal policy* which is guaranteed to achieve at least the lower bound value in any of the MDPs in the group. Although we sacrifice guaranteed convergence to the original optimal solution with this method, Dean et al. showed that their method substantially reduced the model sizes in various experiments [28].

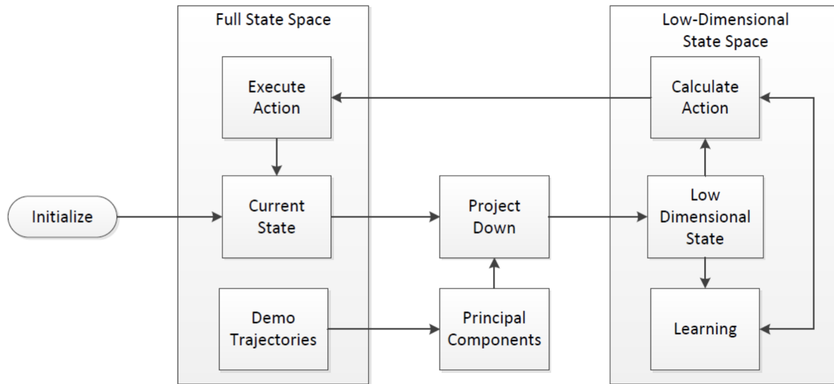

Figure 3.3: Reinforcement Learning with PCA compressed state-space from the work of Curran et al. [31]

There have been other works which have built on top of this. Ravindran et al. investigated abstracting states using homomorphism, which involves mapping a MDP $\mathcal{M}$ into another MDP $\mathcal{M}'$ by removing some details of the system whilst preserving the dynamics of the system [29]. Ferns et al. have also explored multiple different metrics to measure similarity between states, allowing us to aggregate states and reduce the size of the state space [30].

State-space compression using PCA

Principal Component Analysis (PCA) is a mathematical method which compresses data into smaller dimensions, whilst trying to minimise the loss of information [10]. Curran et al. successfully applied this technique to Reinforcement Learning by using PCA to compress a large state space into one with lower dimensionality [31]. Figure 3.3 draws out the flow of their method. In each iteration of the method, the Reinforcement Learning agent starts by projecting its current state to the lower-dimensional state space using PCA. Then, it takes an action which leads it to its next state in the full state space. Through observing the reward, the agent performs a learning update in the lower dimensional state.

They found that using PCA compression allowed the agent to converge to a good policy faster as we operate in a lower dimensional state space. However, it is inevitable that we lose some information whilst performing PCA, so the optimal policy we converge to will always be worse than if we were working with the full state space [31]. Therefore, there is a critical trade-off between convergence and performance with this method. For instance, this method may not be suitable for problems where each feature of the state space is vital to solving the problem, as performing PCA may remove these important features. Secondly, it is also difficult to decide the dimensions of the lower-dimensional state space. If we don't have a good idea of what the problem is, it may be hard to gauge whether we are removing important features or not.

3.4 Summary

After discussing various existing methods which address the state-space explosion problem, we can draw out two important underlying common techniques they use:

- **Simplifying the underlying problem:** State-space compression or abstraction is a means of simplifying the underlying problem into something of a lower dimension. Hierarchical Reinforcement Learning can also arguably be simplifying the problem by decomposing it into several simpler sub-tasks.

- **Distributing computation:** Hierarchical Reinforcement Learning aims to distribute the computation of the problem by defining sub-problems that can sometimes be even solved simultaneously [32]. We've also seen that Multi-agent Learning distributes the learning between several agents, allowing cooperation and more efficient learning.

In our approach, we draw inspiration from parts of the works discussed. We primarily tackle problems which have a sparse transition probability matrix $\mathcal{P}$, meaning that some state transitions rarely occur, making some states 'irrelevant' to other states. Through setting these transition probabilities to zero, we can define a state-space decomposition and simplify the problem into multiple separate Reinforcement Learning problems, each one learning in a different state sub-space. This enables us to distribute the learning between several lower-dimensional neural networks, leading to faster training. We chose to investigate this approach as there are many large Reinforcement Learning problems which possess a sparse transition probability matrix, but to our knowledge, there are currently no methods which exploit this property in order to accelerate training. Our approach is further detailed in chapter 4 and 6.

Chapter 4

A new approach: State-space decomposition in Reinforcement Learning

The approach we investigate in this project was initially proposed as an idea by Leung et al. in [33]. We want to address the state-space explosion problem for large MDP problems with a sparse transition probability matrix $\mathcal{P}$. When $\mathcal{P}$ is sparse, it can be decomposed into multiple sub-matrices by setting some state transition probabilities to zero if it is less than a threshold ϵ (See figure 4.1). When we normalise the resulting sub-matrices, we end up with multiple independent smaller MDPs which can be learned on separate neural networks. Doing this decomposes the full state space of the MDP $\mathcal{S}$ into different smaller sub-spaces $S_1, ..., S_4$ and reduces the dimensions we operate with during training.

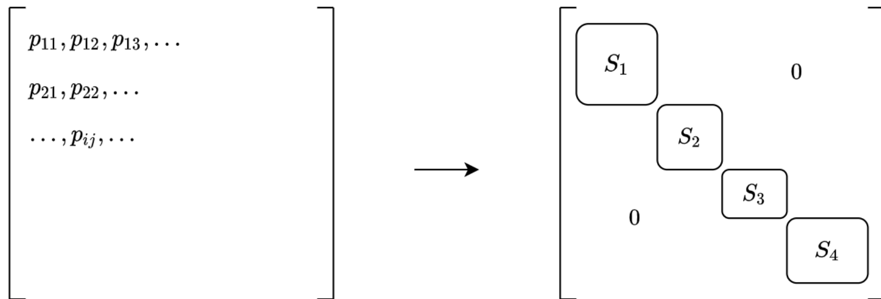

Figure 4.1: Decomposing state space $\mathcal{S}$ into sub-spaces $S_1, ..., S_4$ through simplifying the transition probability matrix $\mathcal{P}$

However, through setting some transition probabilities to zero, we sacrifice accuracy in our learning as we have removed information that could be vital for the optimal solution. Therefore, we ensure that we still learn how to interact between sub-spaces by introducing a combining neural network. This network trains on transitions that take us between different sub-spaces. Figure 4.2 shows a hierarchical neural network architecture that combines everything together. The training will involve two stages:

- **Stage 1**: Each sub-space neural network will be trained using transitions that interact within their corresponding sub-spaces. Their outputs will be the Q-values for their corresponding MDP problem.
- **Stage 2**: The combining neural network will then utilise the learned sub-space Q-values $Q_1(S, A), ..., Q_k(S, A)$ and continue to train on transitions between different sub-spaces to converge to the state-action function for the over-arching problem $Q(S, A)$. At the same time,

the combining neural network could carry out back-propagation to adjust the parameters of the sub-space neural networks.

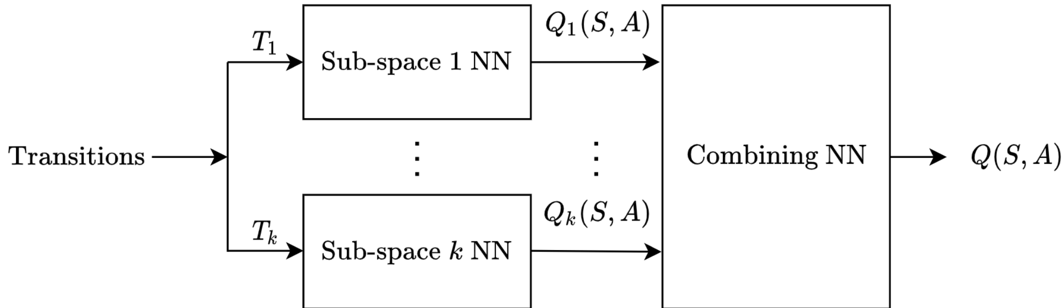

Figure 4.2: Network architecture of our new approach. The sub-space NNs train within each state subspace and the combining NN takes the outputs of the sub-space NNs to learn the complete set of Q-values in the MDP. T_k denotes the experienced transitions which occur within sub-space k.

The combining neural network still operates in the full dimension space. However, since we have sub-space neural networks that operate in a lower dimension space, convergence for those is achieved more efficiently. This allows the combining neural network to work on top of good estimates of the state-action function early on in training.

In this report, we will be developing this idea into a novel method called **State-space Decomposition Reinforcement Learning (SSD-RL)**.

4.1 Challenges

Using state-space decomposition for Reinforcement Learning is only a preliminary idea for now. Developing the idea into a usable method poses several key challenges which we will be solving in our research.

1. **Decomposing a state-space effectively**
   Theoretically, we are able to apply a state-space decomposition by observing the problem's transition probability matrix. However, in real life, we often do not have access to this. Therefore, we must construct a state-space decomposition by exploiting our knowledge of the problem's system dynamics. We will need to explore the ways in which we can do this so that using state-space decomposition in Reinforcement Learning is beneficial.

2. **Developing SSD-RL's training sequence**
   We outlined the two stages of training previously. However, this needs to be refined into a specific algorithm. There are various ways we are able to train. For example, we could train both the sub-space networks and the combining network at the same time, or we could separate the two stages completely and define a point at which we switch from stage 1 to 2.

3. **Ability to converge to the optimal solution**
   As we are splitting up our problem into multiple neural networks, convergence can be a challenge. We need to ensure that our method is able to glue all the learned sub-problems together to form an over-arching solution. In cases where the state sub-spaces are very disjoint, this may be very straight-forward, but when they aren't, this could be a difficult task.

4. **Efficiency of the forward pass**
   As our forward pass is slightly more complex relative to other methods like Deep Q-Learning, we need to ensure that we are able to implement SSD-RL in an efficient way so that our performance gain is not invalidated by it.

Chapter 5

Developing SSD-RL using grid-world environments

5.1 Grid-world environments

In order to develop our method and investigate its performance, we firstly need to find a suitable Reinforcement Learning environment we want to test on. We chose to use grid-world environments as they were simple and customisable. The optimal policy at each grid position is also relatively easy to figure out, which helps us in analysing how well our Reinforcement Learning agent is learning.

As we aim to decompose the state space into several disjoint ones through approximation, we ideally want an environment where we can define different sub-spaces and where there is very little interaction between them. We decided to create a grid-world maze environment with two rooms as shown in Figure 5.1. The dimensions of the grid is 35x35. The goal of this environment is for the agent to cross to the other room and reach the goal state. There are four possible actions the agent can take: up, right, down, and left. The state will be represented as the 2D coordinate of the agent's current position. With this environment, we can split the state space into two sub-spaces, one being the group of state coordinates in the first room, and the other being the group of state coordinates in the second room. Since there is only a small gate leading into the second room, there is a small probability that the agent crosses state sub-spaces. The state-space decomposition in this environment is relatively simple and visual, which allows us to understand the results of our experiments more easily.

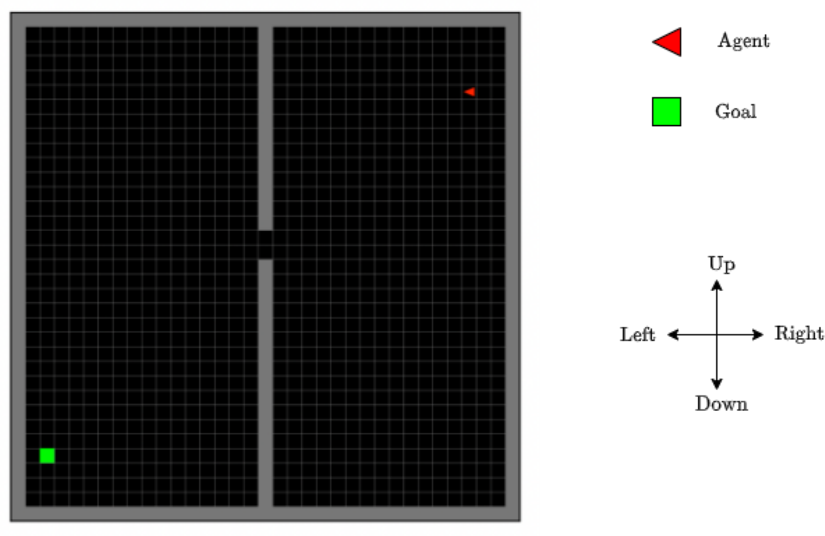

Figure 5.1: Room maze environment

We created two variations of this environment's reward function in order to assess the perfor-

mance of our approach. These environments were implemented on top of an open-source reinforcement learning environment library named 'Minigrid' [34].

Variation 1: Sub-goal Room Maze

In this variation, we will test an environment where the initial room's reward function will guide the agent towards the opening of the next room. The next room's reward function will then guide the agent towards the final goal. This way, the agent is able to learn a useful part of the optimal solution when training in a decomposed manner as the reward system has underlying sub-goals defined.

Let s_0 denote the agent's starting state. The reward function taking in state s is as follows:

$$r(s) = \begin{cases} \|gate_pos - s_0\|_F - \|gate_pos - s\|_F\,, & \text{if } s \in \text{initial room} \\ \|goal_pos - s_0\|_F - \|goal_pos - s\|_F\,, & \text{otherwise} \end{cases} \tag{5.1}$$

Variation 2: Single-goal Room Maze

This is a more straight-forward version of the environment, but could possibly be more difficult for SSD-RL to solve. The reward function only accounts for the agent's distance from the final goal. Unlike variation 1, it doesn't guide the agent through the gate. Therefore, the local optimal policy we learn in the initial sub-space (initial room) will not be part of the global optimal policy when we merge the two sub-spaces together. This reward function will be able to test if our approach easily gets stuck in a local optimum. It will also help test if our combining neural network is able to connect the two sub-spaces together well and form the global optimal policy.

Let s_0 denote the agent's starting state. The reward function taking in state s is as follows:

$$r(s) = \|goal_pos - s_0\|_F - \|goal_pos - s\|_F \tag{5.2}$$

5.2 Initial experiments

Training loop

We conducted experiments with two different implementations of SSD-RL's training loop on the above environments.

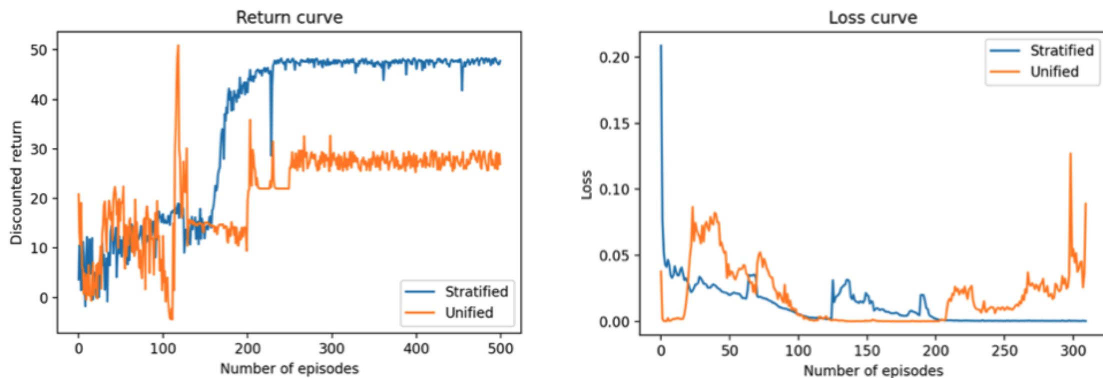

Figure 5.2: Comparison of loss curves between SSD-RL with stratified learning and SSD-RL with unified learning.

1. **SSD-RL with unified learning:** This implementation involved training the sub-space networks and the combining network together. One forward pass would involve forwarding a transition to its corresponding sub-network and directly into the combining network. Backpropagation would then alter the weights of both the corresponding sub-network and the

combining network. We found that this method performed poorly as it rarely reached convergence to the optimal policy. The loss curve in figure 5.2 also showed large fluctuations, instead of a stable downward sloping curve. It is difficult to detect the exact cause of this, however, we suspect that it is because the combining network is constantly being adjusted based on different sub-networks which are also constantly changing, resulting in learning that may 'conflict' with each other.

2. **SSD-RL with stratified learning:** This implementation involved training the sub-space networks and the combining network separately. Hence, we introduced an additional hyper-parameter N, which decides the point at which we switch from training the sub-networks to the combining network. When we train the combining network, the sub-networks' weights aren't updated and are essentially frozen. We found that this method performed much better as it was able to converge to the optimal solution. We inspected the each sub-network and also found that their learned Q-values reached a good estimate before training the combining network. The loss function in figure 5.2 also showed a clear downward trend, showing that our network was learning in a stable manner.

Transition replay buffer

Drawing inspiration from Deep Q-Learning proposed by Minh et al., we adapted their concept of a transition replay buffer to create one for our method as well [11]. We found that the replay buffer helped the learning immensely. It sped up training and helped us make more efficient use of our past experience. In our environments, it is very rare to encounter the gate. However, with the replay buffer we were able to retain that encounter for a longer period of time, allowing us to 'revisit' the gate state and learn more efficiently. See section 6.3 for more technical details about our adapted replay buffer.

Exploration policy during training

We implemented an ϵ-greedy exploration policy during training (See equation 2.8). This enabled us to balance exploration and exploitation. We experimented with different methods of decaying ϵ:

- **Exponential decay:** We found that exponential decay was the least effective, as it decayed too quickly. This caused us to exploit our learned knowledge too early on, preventing us from learning the optimal solution. Our learned policy would often end up leading the agent to the bottom-left hand corner of the first room, which is the local minimum solution. It rarely explored enough to even encounter the gate into the second room.

$$\epsilon = \frac{1}{\text{training episode number}} \tag{5.3}$$

- **Linear decay:** Linear decay worked much better as it managed to converge to the optimal solution. There was sufficient exploration for us to encounter the gate and to learn that it lead to the goal.

$$\epsilon = 1 - 0.01(\text{training episode number}) \tag{5.4}$$

- **Double-start linear decay:** We tried to improve upon the linear decay by introducing double-start linear decay. Since we had two stages of training in SSD-RL, we observed that the performance slightly dipped when we switched from training the sub-networks to the combining network. This was because we were using a newly initialised network to make decisions. In order to prevent over-exploiting at this inflexion point, we set ϵ to a higher value before continuing to decay it linearly. See equation 5.5 and figure 5.3. This slightly reduced the time taken for our network to recover from the performance dip.

$$\epsilon = \begin{cases} 1 - 0.01(\text{training episode number}), & \text{if training epsisode number} < N \\ 0.7 - 0.01(\text{training episode number} - N), & \text{otherwise} \end{cases} \tag{5.5}$$

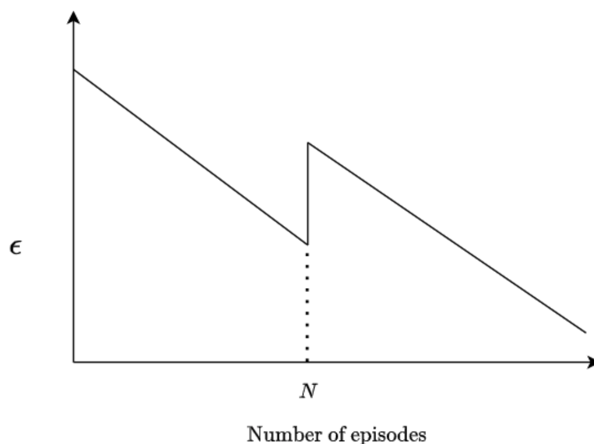

Figure 5.3: ϵ value during training with Double-start linear decay

5.3 Performance evaluation

We will be using the state-of-the-art Deep Q-Learning (DQL) method described in section 2.3.4 as our baseline method. We performed tests on both environment variations and compared DQL's performance with our method. In order to carry out a fair comparison, the network architectures in both methods contain a very similar amount of parameters. They also utilise a very similar learning rate and batch size during training. Details about both method's hyper-parameters can be found in appendix A.

5.3.1 Reward curves

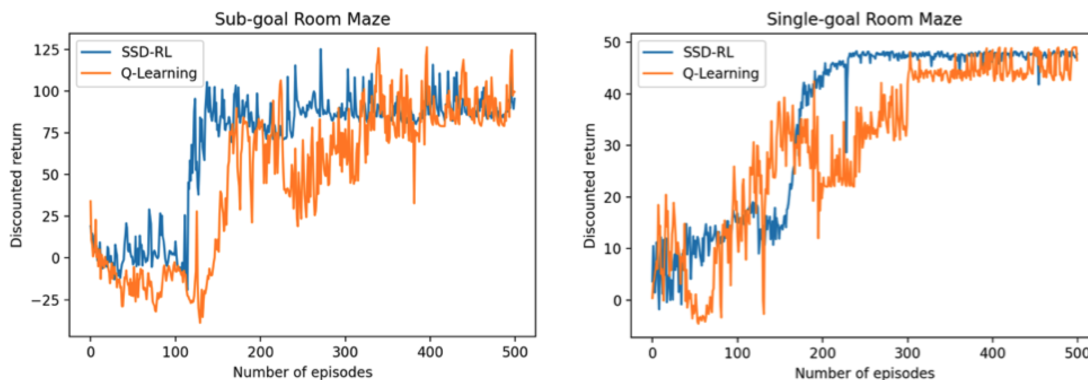

Figure 5.4: Comparison of return curves between SSD-RL and Deep Q-learning for the two variations of the room maze environment. Results are averaged over 3 runs.

Figure 5.4 compares the two method's performance on our gridworld environments. We can observe that SSD-RL has better stability during training. DQL's performance tends to fluctuate more before it manages to converge, whilst SSD-RL presents a smoother curve where the point of convergence is easily found. We can also see that SSD-RL clearly converges faster than DQL. In the sub-goal room maze environment, SSD-RL converges at the 115th episode, compared to DQL which converges at the 310th episode. SSD-RL achieves around a 60% reduction in number of training episodes. In the single-goal room maze environment, SSD-RL achieves around a 25% reduction in number of training episodes.

The reason why SSD-RL performs much better in the Sub-goal room maze is because the reward system guides the agent towards the gate. Therefore, our first sub-network, which learns about

states the initial room, learns part of the global optimal policy even before we train the combining network. Conversely, with the single-goal room maze environment, our first sub-network would not have any knowledge of the future rewards we are able to obtain when crossing the gate, therefore it would initially learn how to arrive at the closest state to the goal within the initial room. This means that we would need more training time to correct this policy when training the combining network.

5.3.2 Effects of changing the state-space size

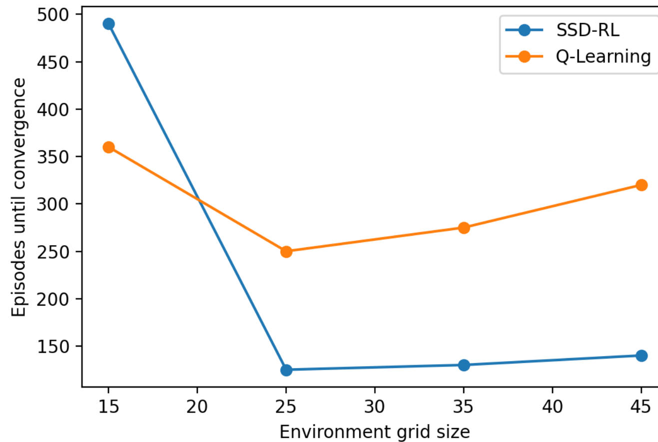

Figure 5.5: Performance of SSD-RL and DQL for different grid sizes in the sub-goal room maze environment

We ran experiments for different grid sizes in order to see how the state-space size affected SSD-RL's performance. In each run, we adjusted SSD-RL's hyper-parameter N to ensure that we didn't over-train on smaller environments and under-train on larger ones. In figure 5.5, we see that SSD-RL performs poorly for small state spaces, but performs much better than DQL when the grid size is larger than 20. The performance gain of using SSD-RL relative to DQL gets more apparent as the state-space size increases. These results suggest that for smaller problems, it's more inefficient to split up training into multiple smaller networks as one larger network is sufficient to learn the simpler Q-value function quickly.

Chapter 6

SSD-RL: State-space Decomposition Reinforcement Learning

Through experimenting with grid-world environments and tweaking our method throughout, we developed SSD-RL, a Deep Reinforcement Learning method which utilises state-space decomposition to exploit environment dynamics in order to speed up training. This chapter presents the technical details of our method.

6.1 Network architecture

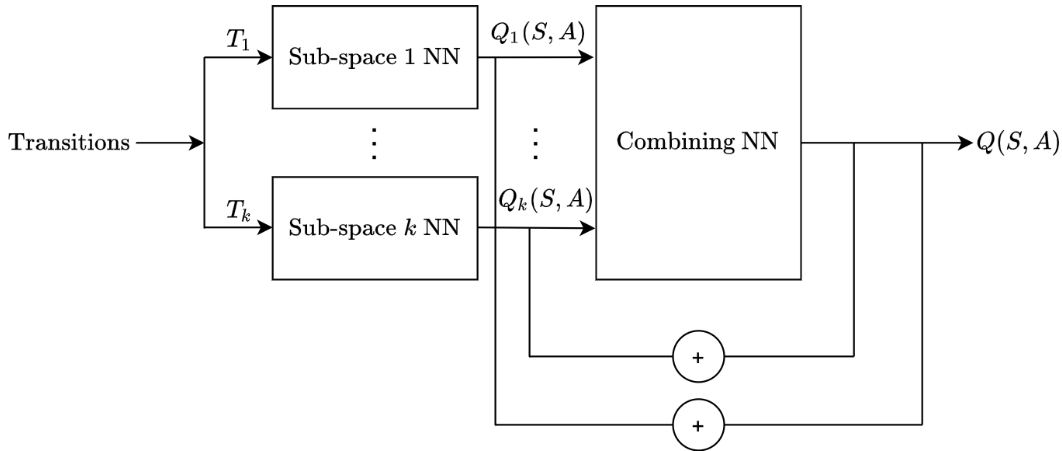

Figure 6.1: SSD-RL Network architecture. The + connections represent skip connections.

Figure 6.1 shows the network architecture of SSD-RL. When using SSD-RL, we require a state-space decomposition which is created by the programmer. This is created by splitting the full state space up into two or more smaller sub-spaces. For k number of sub-spaces, there are k sub-space networks which are responsible for learning the Q-values for its corresponding subset of states. The inputs to each sub-space network are transitions which travel within their corresponding state sub-space. For example, in figure 6.1, T_k denotes a transition where the current state and next state are both in the kth sub-space.

The outputs (Q-values) of these sub-space networks are then passed as input into the combining neural network which aims to refine the input Q-values to its true value by taking into account inter-subspace transitions. There are additional skip connections which add the inputs of the combining network to the output of the combining network. We added these to speed up learning.

Since the combining network takes in Q-values from the sub-space networks and tweaks them, its output should be very similar to its input. With the skip connections, our combining network only needs to learn the difference between the Q-values learned in the sub-space networks and the true Q-values. This way, we no longer need to learn the identity function, which may take a long time if environments have large reward values.

In our grid-world environments, the probability of crossing our defined state sub-spaces was very small. Hence, we chose to use a combining network with less parameters than a single sub-space network. This worked well as there was more to learn within each sub-space than between the two sub-spaces. There are only two transitions which can cross sub-spaces - one which takes the agent from the gate position to second room, and another which takes us from the gate back to the initial room. However, with different environments, the number of parameters in each network can be changed. For our activation functions, we chose to use ReLU between each linear layer. ReLU helps alleviate the vanishing gradient problem, and it's also very fast to compute [9]. Further details about each layer's hyper-parameter values for each neural network can be found in appendix A.

6.2 Training

There are two stages of training in SSD-RL. We start in stage 1, and after N (hyper-parameter) episodes, we transition to stage 2. During training, our agent explores the environment using a double-start decay ϵ-greedy policy as described in equation 5.5.

6.2.1 Stage 1: Training within state sub-spaces

Stage 1 of training aims to learn the Q-values of each state sub-space. By the end of this stage, the sub-space neural networks should be able to learn a local optimal policy within its own sub-space, which will then be used to find the global optimal policy in stage 2 of training. During stage 1, we only learn from transitions that travel within the same sub-space. The pseudo-code is shown in algorithm 2. At this stage, we only update the weights of the sub-space neural networks.

Algorithm 2: SSD-RL: Stage 1 of training

```
for k in num_subspaces do
    Initialise a Q-network Q_k with random weights θ_k;
end
for each episode do
    S ← S_init;
    for each step in episode do
        Choose action A from current state using ϵ-greedy policy from Q_k;
        Take A and observe reward R and next state S';
        Store transition (S, A, R, S') in decomposed replay buffer;
        Sample mini-batch B;
        for k in num_subspaces do
            θ_k ← θ_k − (1/B_k) Σ_{b_k} ∇_{θ_k} [r + γ max_a Q_k(S', A) − Q_k(S, A)];
        end
        S ← S';
    end
end
```

▷ ∇_{θ_k} denotes the gradient w.r.t. θ_k

▷ B_k denotes the sub-batch of transitions which travel within sub-space k

6.2.2 Stage 2: Training across state sub-spaces

After N episodes, we enter stage 2 of training. The main aim of this stage is to combine our learned sub-problems to form a global optimum policy for the full environment. We do this by refining our learned Q-values from stage 1 by taking into account inter-subspace transitions. At each step in an episode, we train on both within sub-space transitions and inter sub-space transitions. However,

now, we only update the weights of the combining neural network, and the sub-space network weights are frozen.

Let C denote the combining neural network with weights θ_C. C takes in an array of Q-values for a specific state, $Q(S, A)$. It then outputs the refined Q-values for that state, which we will denote as $C(Q(S, A))$. The training loop is very similar to stage 1, but the weight update rule and exploration policy is different. For a transition which travels from sub-space k to j, the weights of the combining neural network are updated as shown in line 14 of algorithm 3. The exploration policy is shown in lines 6-10. As we enter into stage 2, we will encounter a slight performance dip during training as we are bringing in a newly initialised neural network. To diminish this effect, we continue to use the sub-space networks to obtain our exploration policy for T (hyper-parameter) episodes. This gives us time to improve the combining network to a suitable level before we use it to explore the environment. After T episodes, we start choosing actions from the combining network C instead of Q_k.

Algorithm 3: SSD-RL: Stage 2 of training

Initialise combining network C with random weights θ_C;
for ***each*** *episode* **do**
  $S \leftarrow S_{init}$;
  $k \leftarrow$ get_subspace(S);
  for ***each*** *step in episode* **do**
    if *episode* $\leq T$ **then**
      Choose action A from current state using ϵ-greedy policy from Q_k
    else
      Choose action A from current state using ϵ-greedy policy from C
    end
    Take A and observe reward R and next state S';
    Store transition (S, A, R, S') in decomposed replay buffer;
    Sample mini-batch $\mathcal{B}$;
    $\theta_C \leftarrow \theta_C - \frac{1}{B}\sum_b \nabla_{\theta_C}[r + \gamma \max_a C(Q_j(S', A)) - C(Q_k(S, A))]$;
    $S \leftarrow S'$;
    $k \leftarrow$ get_subspace(S);
  end
end

▷ j is the sub-space which the transition travels towards
▷ k is the sub-space which the transition travels from

6.3 Decomposed replay buffer

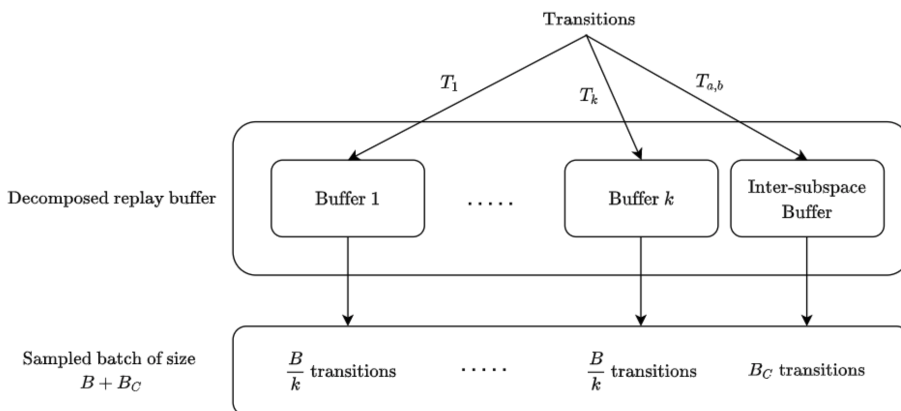

Figure 6.2: Sampling method of the decomposed replay buffer for batch size $B + B_C$

We created a decomposed replay buffer in order to store past experience and make our SSD-RL implementation more efficient (See figure 6.2). A basic replay buffer stores all its past experience in a queue, and samples random batches from this buffer at each step of training. With SSD-RL, our transitions are categorized by which sub-space they operate in, and whether or not it stays within a sub-space or crosses to another one. Not only does this storage structure allow us to implement the most efficient forward pass of SSD-RL, but it also allowed our sampling method to improve SSD-RL's learning stability.

The decomposed replay buffer contains an individual buffer for each sub-space. Transitions which travel within subspace k, would be stored in Buffer k in figure 6.2. Transitions which travel between two different sub-spaces are stored in the inter-subspace buffer. Through splitting up the agent's experience into separate buffers, we are able to pre-sort our samples to enable a more efficient sampling implementation.

When sampling a batch of transitions for learning, we obtain an equal amount of transitions in each sub-space buffer. For example, If our batch size was 256 and we had 4 state sub-spaces, we would sample $256/4$ transitions from each sub-space buffer. This design decision was made as we wanted to guarantee that a minimum number of transitions were sampled per subspace. If we didn't do this, we could be updating large neural networks based on only a few samples, leading to unstable training. Next, we sample B_C transitions from the inter-subspace buffer, which contains inter-subspace transitions. We chose to allow the programmer to specify how many inter-subspace transitions to sample as it really depends on how often the agent crosses sub-spaces in its environment. If crossing sub-spaces is common, B_C should be larger, and if not, B_C should be smaller. As a point of reference, for our Room Maze environment in section 5.1, B_C was set to 5. Finally, all of these sampled transitions are concatenated together to form one batch of size $B + B_C$.

Chapter 7

Decomposability factor for SSD-RL

A big question still remains about SSD-RL: how disjoint do our state sub-spaces need to be in order for it to perform well? In this section, we investigate the state-space 'decomposability factor' needed in order for SSD-RL to succeed.

7.1 Definition

We define the decomposability factor as the extent at which a state-space can be split into separate sub-spaces. It is a real number between 0 and 1. The higher the decomposability factor, the more disjoint the sub-spaces are, meaning that the agent is most likely to interact within its own current state sub-space. The lower the decomposability factor, the more likely that inter-subspace interactions will occur in the environment, making the state sub-spaces less detached from each other. In the extreme cases, when the decomposability factor is 1, we can basically split up our Reinforcement Learning problem in two completely separate ones as there are no interactions between the state sub-spaces. When the decomposability factor is 0, the interaction between different sub-spaces is as common as ones within each sub-space.

7.2 'Two goals' environment

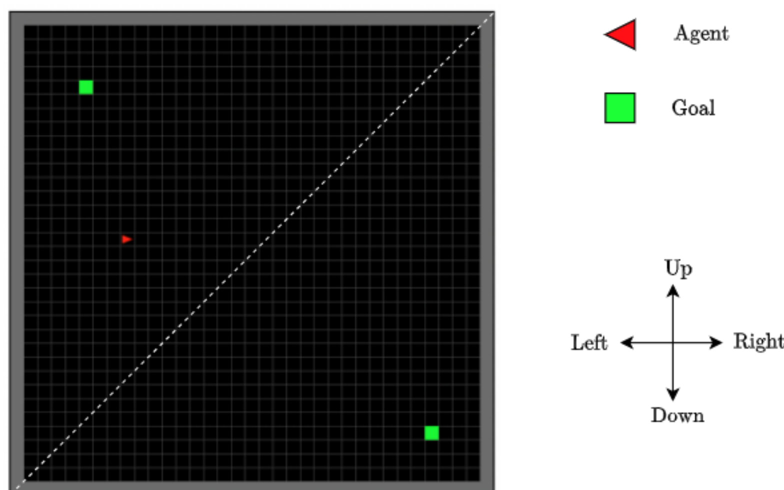

Figure 7.1: Two goals environment. The dotted line indicates the split between the first state sub-space and the second one.

In order to quantitatively analyse the effects of the decomposability factor on the performance of SSD-RL, we created an environment which allowed us to easily adjust the decomposability factor of the state-space.

We created the 'Two goals' environment, which is shown in figure 7.1. There are two state sub-spaces which are shown by the dividing dotted line in the figure. In each initialisation of the environment, the agent starts in a random position and gets a random task assigned to it. The task could either be to travel to the closest goal to the agent's starting position, or to the further goal, which requires the agent to cross state sub-spaces. The task assignment probability is adjusted by an ϵ value. When ϵ is close to zero, there is very little probability that we get assigned the further goal. When ϵ is close to 1, there is a high probability that we get assigned the further goal. By changing the ϵ value in our experiments were are able to change how often we cross state sub-spaces, thereby changing the decomposability factor of the environment.

The states, actions and rewards of this environment is as follows:

- **States:** The state consists of the agent's current position, and the task it got assigned. The position is a 2D coordinate, and the task is a binary number. These two components are concatenated together to form the state. Task 0 indicates that the agent needs to find the path to its closest goal, task 1 indicates that the agent needs to find the path to the further goal, crossing sub-spaces as a result.
- **Actions:** There are four possible actions: up, right, down left.
- **Rewards:** The reward function is the distance between the agent and the assigned goal.

7.3 Effects on performance gain

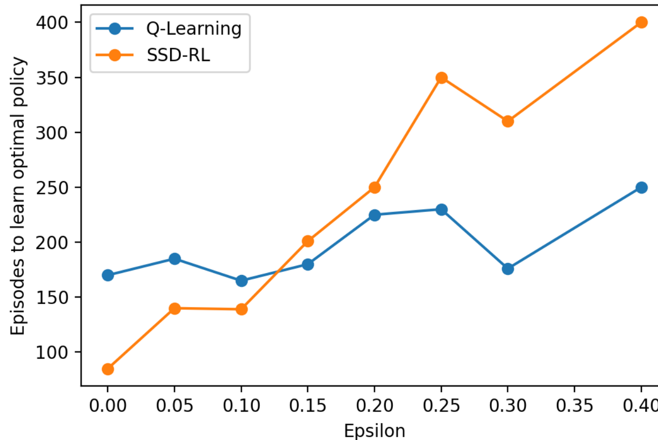

Figure 7.2: Performance analysis for different ϵ values in the Two goals environment. Each point is averaged over 5 runs.

Similarly to our previous experiments, we compared SSD-RL's performance to Deep Q-Learning (DQL). We ran experiments with different values of ϵ and compared the number of episodes it took for each method to converge to the optimal policy. By oberving figure 7.2, we see that **SSD-RL outperforms DQL as long as ϵ is smaller than 0.125.** SSD-RL's performance gain is most significant when ϵ is very small (or when the decomposability factor is high). This is due to the fact that most interactions occur within the state sub-spaces, meaning that most of the Q-values we learn in stage 1 of training are already very accurate. As we enter stage 2 of training, there is not much need in altering these values drastically.

When ϵ is greater than 0.125, SSD-RL's performance continues to decrease. It performs at a similar level to Deep Q-Learning when ϵ is between 0.15 and 0.20. However, SSD-RL's performance significantly worsens for large ϵ values (or when the decomposability factor is very low). Since there are a lot of interactions between our state sub-spaces when the decomposability factor is low, the Q-values we learn in stage 1 of SSD-RL's training will not be accurate, and so our learned optimal policy will need a lot of correction in stage 2 of training. As SSD-RL's network architecture is

designed to take advantage of disjoint state sub-spaces, it makes sense that it is not as effective when the environment's state space is less decomposable.

In summary, there is a strong relationship between SSD-RL's performance and the decomposability factor of our state sub-spaces. To maximise our gains, we should be aiming to create a state-space decomposition where the sub-spaces are as disjoint as possible. Also, there comes a point when our decomposability factor is so low that we would be better off using other Reinforcement Learning methods.

Chapter 8

Applying SSD-RL to Alibaba's dataset

So far, we've shown that SSD-RL outperforms our baseline method in gridworld environments with the appropriate decomposability factor. As we want SSD-RL to be used in real-life applications, we now investigate its performance on a real-life data set. We chose to create an environment involving two data centers, each modelled by Alibaba's cluster trace data set, which contains data about the characteristics of modern data centers and their workloads [35]. The problem which the Reinforcement Learning agent aims to learn is how we can distribute incoming workload to the data centers in a way that is feasible and least costly. We created two data centers so that we could apply state-space decomposition to the problem. We split up the problem into two: one which aims to allocate workload arriving at one data center, and the other aiming to allocate workload arriving at the other center.

8.1 Data processing

The trace data set includes information about 4000 machines over a period of 8 days. It documents each machine's CPU utilisation percentage, memory utilisation percentage and many more metrics. To simplify the large data set, we only utilised the CPU utilisation percentage column of the data. We also chose to only use the data for 10 machines.

We chose to use a sub-set of time-steps to create our environment. We extracted one of the busier hours in the data-set to ensure that workload distribution wasn't a trivial solution, forcing the agent to actually learn how to serve incoming workload with limited resources. Finally, we extracted the CPU utilisation time series for each of the 10 machines so it could be readily used for simulation in our environment.

8.2 Creating the 'Workload Distribution' environment

Our environment consists of two data centers, each with 5 operating machines. The CPU utilisation percentage of these machines are modelled by the time series we extracted from our data set. At each time-step, a random amount of workload is sent to **one** data center which needs immediate service. The agent is responsible for deciding where to distribute incoming workload which arrive at either data center. The reward system is constructed to incentivize feasible and efficient allocation of workload.

Arrival of workload

At each time-step of the environment, we need to decide which data center the new workload arrives at. We show our algorithm in 4.

At the first time-step, we choose a random data center. Each step after that, we sample a random number $x \in [0, 1)$. The workload continues to arrive at the initial data center until x is smaller than a defined constant $\epsilon \in [0, 1)$. When this happens, our workload starts arriving at the other data center. Workload then continues to arrive at the other data center until we encounter

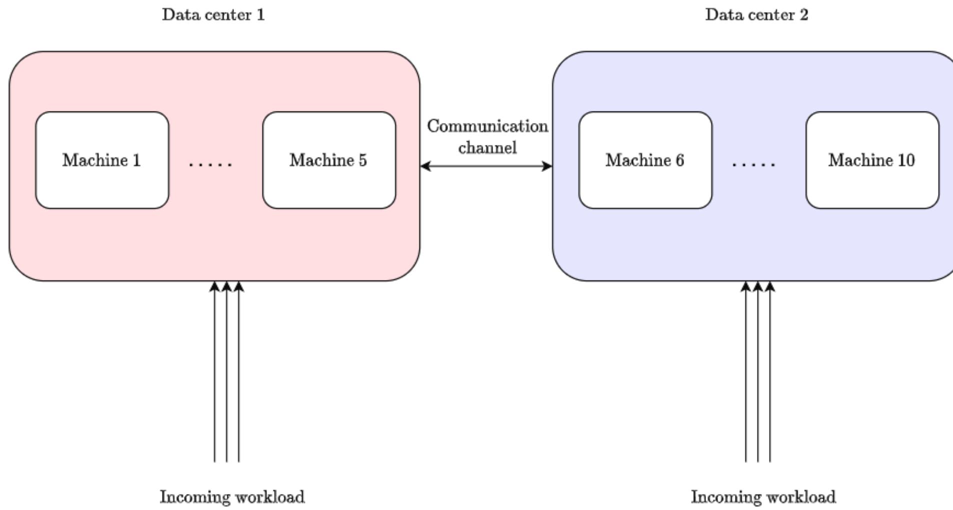

Figure 8.1: Workload Distribution environment

another sampled random number that is smaller than ϵ. To illustrate the effects of ϵ, let D1 and D2 be data centers 1 and 2 respectively. When ϵ is close to 0, the workload arrivals would look something like D1, D1, D1,...D2, D2, D2... When ϵ is close to 1, the workload arrivals would look like D1, D2, D1, D2...Essentially, ϵ acts as a toggle between the two data centers. We chose to define ϵ so that we are able to control the decomposability factor of the environment if we were to split the state sub-spaces by data center.

Algorithm 4: Workload arrival

```
datacenter ← 0 ;
for each step in episode do
    r ← random();
    cpu_requirment ← random() * max_cpu_utilisation ;
    if r ≤ ϵ then
        datacenter ← not datacenter
    end
    Send cpu_requirment amout of workload to datacenter ;
end
```

States

Our state consists of the current CPU utilisation of each of the 10 machines, and information about the current workload arriving at the data centers. Below is an example of a state:

`[cpu1, cpu2, ..., cpu10, workload_requirement, data_center]`

The first 10 integers are the CPU utilisation percentage values for the 10 machines at the specific time-step of the environment. The first 5 CPU utilisation values are ones from the data center 1, and the next 5 are ones from data center 2. Next, there is a workload requirement which indicates the amount of CPU utilisation percentage it needs to be completed. Lastly, the data center element indicates which data center the workload is arriving at.

Actions

1. **Allocate to local machines:** This action allocates the full workload into an available machine in the agent's local data center. If there isn't enough CPU resources, it partially fulfills the workload and the rest of the workload is not served.

2. **Allocate to remote machines:** This action allocates the full workload into an available machine in the agent's remote data center. Similarly, if there isn't enough CPU resources, it partially fulfills the workload and the rest of the workload is not served.

Rewards

If the agent successfully serves the full incoming workload at its local data center, it receives the maximum reward. If the agent successfully serves the full incoming workload in the remote data-center, it still receives a positive reward signal, but a smaller one as there is a communication and time cost when sending work to a remote machine.

In the case where all machines are at full capacity, the agent can serve the workload partially. The reward signal for this situation is variable depending on how much of the original workload the agent managed to serve. However, the reward is halved if it is served in the remote data center. This ensures that we still favour partially serving a request locally versus if we were to do it remotely.

8.3 Evaluation of results

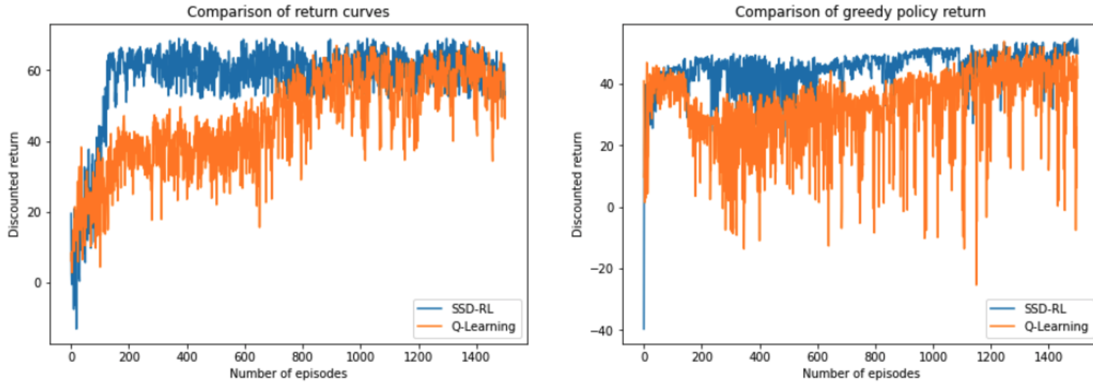

Figure 8.2: Performance comparison between SSD-RL and DQL with the Workload Distribution environment with $\epsilon = 0.2$

During training, we split up the environment's state space by data center. Therefore, if requests arrive at the first data center, this will be considered as a transition in sub-space one, whereas if requests arrive at the second data center, transitions are in sub-space two. We chose a higher decomposability factor for our environment as we learned that that is when SSD-RL performs best. We set ϵ to 0.2.

Figure 8.2 compares SSD-RL and DQL in two ways: by their return curves and their greedy policy return curve. The greedy policy return curve measures the total discounted return the agent receives when executing the best possible policy according to the neural network at that specific training episode. We ran SSD-RL and DQL with the same random environment seed to ensure a fair comparison.

We can observe that SSD-RL outperforms DQL in both curves. SSD-RL is able to reach a high return much earlier on in training, and its greedy policy learns steadily and maintains a higher return than DQL's greedy policy throughout training. The performance gain we get from using SSD-RL is much more significant than the gain we acheived with our gridworld environments. This is due to the fact that we are dealing with a much larger state space, as we saw that the larger the state space, the more beneficial it was to use SSD-RL in section 5.3.2.

8.4 Effect of varying the decomposability factor

We ran SSD-RL for different ϵ values to see the effects of the decomposability factor. Figure 8.3 shows the results. We can observe that the higher the ϵ value (or the lower the decomposability

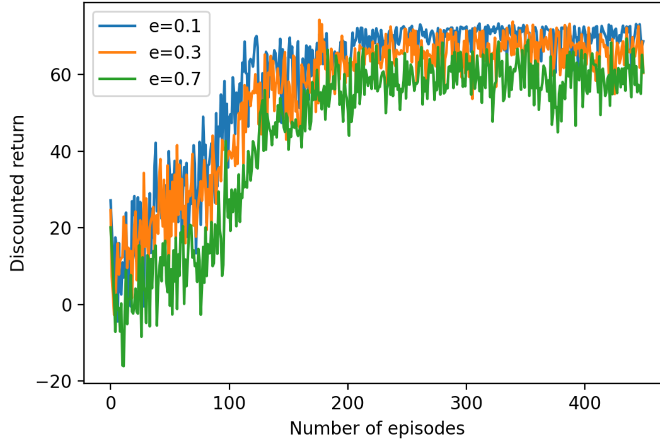

Figure 8.3: Effect of ϵ on performance of SSD-RL. Results are averaged over 5 runs.

factor), the worse SSD-RL performs. For $\epsilon = 0.7$, the curve shows slower learning in the first 100 episodes and it also converges at a sub-optimal policy within 400 episodes. These results are expected as they align with those we got in the decomposability factor experiment in section 7.3. When ϵ is close to zero, there is very little chance that we cross sub-spaces in the environment. Therefore, stage 2 of training takes less time to converge as we have already learned a decent policy in stage 1. When ϵ is close to 1, almost all the transitions the agent experiences crosses state sub-spaces. This means that the majority of the knowledge needs to be learned in stage 2. Stage 1 is almost redundant as we don't have many transitions to train on. Therefore, it learns much slower as we need to correct a lot of Q-values we learn in stage 1 during stage 2.

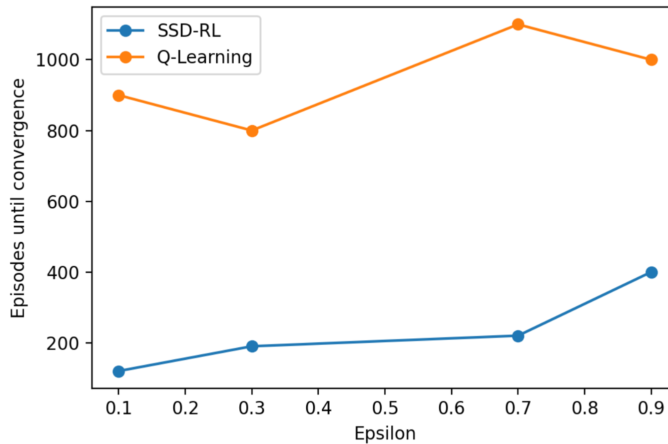

Figure 8.4: Comparison of convergence time between SSD-RL and DRL for different values of ϵ in the 'Workload Distribution' environment.

Next, we attempted to find a decomposability factor threshold at which SSD-RL outperforms DQL in our environment. Figure 8.4 shows our results. As expected, SSD-RL's performance slightly dips as we increase ϵ, however, it is an insignificant loss if we look at the bigger picture. Most importantly, we observe that **SSD-RL outperforms DQL for all the ϵ values we tested**, which is surprising but very impressive. After much investigation, we concluded that this was due to the massive increase in our state-space size. We saw that when we had fewer machines in our data centers, DQL would perform much better and the performance difference would not be as drastic. Therefore, having a large state-space makes SSD-RL an attractive method to use. Even when the decomposability factor is low, it can still outperform the state-of-the-art.

Chapter 9

Applicability of SSD-RL

This chapter evaluates the practicality of using SSD-RL for other Reinforcement Learning problems. We discuss several factors which make SSD-RL an effective method to use, and how it compares with other state-of-the-art Reinforcement Learning methods.

9.1 Suitable environments

When using SSD-RL, we need to be aware of our Reinforcement Learning environment. Some environments work well with SSD-RL, whereas others do not.

9.1.1 Feasibility of state space decomposition

SSD-RL relies on an underlying state space decomposition in the Reinforcement Learning problem. This needs to be manually formulated by the programmer. If we don't have sufficient understanding of the system dynamics of our environment, it's difficult to find a good decomposition of the state space which allows SSD-RL to speed up training.

9.1.2 Decomposability factor

The decomposability factor of the environment's state space plays a huge role in the performance of SSD-RL. The higher the decomposability factor or the more disjoint our state sub-spaces are, the better SSD-RL performs. SSD-RL's network architecture is designed to exploit state decomposability and hence will work best under that scenario. However, it's difficult to define an exact value of the decomposability factor which enables SSD-RL to be worth using. We saw that with our Workload distribution environment, changing the decomposability factor did not affect the performance as much as it did with the Two goals environment.

9.1.3 Size of state-space

As we saw in figure 5.5, the state-space size greatly affects the performance of SSD-RL. It is only worth using SSD-RL when we have a large state-space. In addition, we saw that the larger the state-space size, the more performance gain we get from using SSD-RL. With small state-spaces, the two-staged training is an overhead as training everything together would be more efficient.

9.1.4 Reward function

When we developed SSD-RL with gridworld environments, we found that having a reward function which helps the agent learn useful sub-goals in the state sub-spaces improves the performance. This helps because after stage 1 of training, our learned optimal policies in each of the sub-spaces networks are likely to be a part of the optimal global policy when we combine the sub-spaces together. Therefore, the combining neural network doesn't need to alter them too much. Although this is not a requirement for using SSD-RL, it is something we can look out for in order to gain even better performance.

9.2 Hyper-parameters to tune

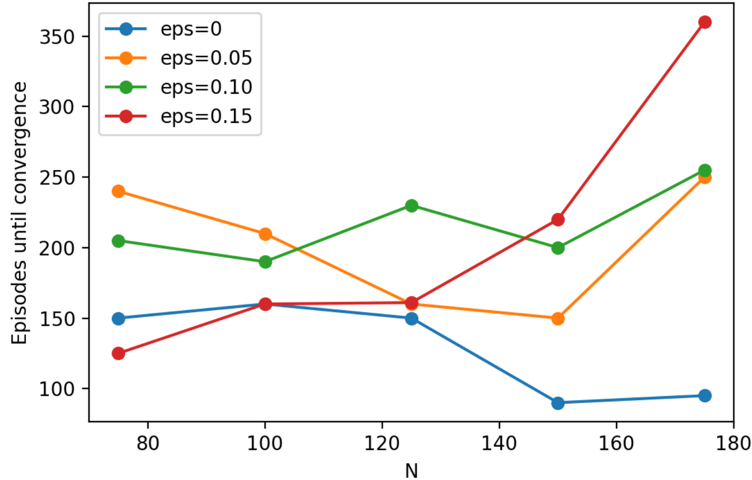

Figure 9.1: Effect of changing N on SSD-RL's performance. We ran this experiment on the Two goals environment with varying ϵ values.

- N: N is the number of episodes we train for before we transition to stage 2 of training. We conducted an experiment to see how N affected the performance of SSD-RL on the Two goals environment in figure 9.1. We can observe that for $\epsilon = 0$ (high decomposability factor), a larger N improves the performance. In this case, there is little interaction between sub-spaces so the knowledge we learn in the sub-space networks can directly lead us to the optimal policy. With a large N, we focus more on training within each sub-space and so it's logical that a larger N would work better for these types of problems. On the other hand, for $\epsilon = 0.15$ (lower decomposability factor), a smaller N works better. This is due to the frequent interactions between sub-spaces which can only be learned in stage 2 of training. Therefore, staying in stage 1 for too long will likely be redundant training as we will need to learn from the inter sub-space transitions to further improve our policy. With ϵ values between 0 and 0.15, the trend is less clear. In this case, N would probably need to be chosen through a random hyper-parameter search.

- T: T is the number of episodes after stage 1 of training where we continue to use our greedy policy from the sub-space neural networks. T is used to prevent a large performance dip when transitioning from stage 1 to stage 2 of training. With our experiments we used values between 10-20. We tuned this hyper-parameter by observing the return curves during training. When T is not tuned well, you can observe a large dip in the return at N episodes, but when T is tuned, this will disappear or become less apparent (See figure 9.2 for comparison).

- **Learning rates:** In SSD-RL, you can have two different learning rates for the sub-space neural networks and the combining neural network. In our experiments, we often used a very similar learning rate for both.

- **Batch sizes B and B_C:** For k state sub-spaces, we sample $\frac{B}{k}$ transitions for each sub-space neural network. If we have a lot of sub-spaces, our batch size B can't be too small as our batch size for each sub-space network $\frac{B}{k}$ would get smaller and smaller. It is important to keep $\frac{B}{k}$ at an appropriately large value to ensure than we are not updating a whole neural network with only a few samples, which could over-write previously learned knowledge, and make training more unstable. B_C indicates how many inter-subspace transitions we want to sample. B_C is usually tuned depending on the decomposability factor of the environment. If inter-subspace transitions often happen, B_C should be larger to reflect that. On the other hand, B_C should be smaller when inter-subspace transitions rarely happen.

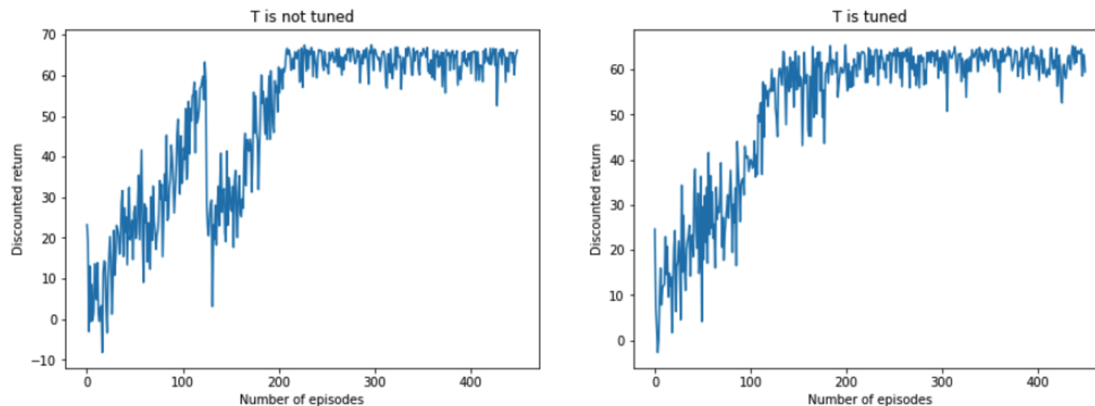

Figure 9.2: Difference between a tuned T value ($T = 15$) and a non-tuned T value ($T = 0$)

9.3 Ease of use

SSD-RL requires the programmer to carry out manual state-space decomposition to simplify the problem. This can be easy or difficult depending on the problem setting. If we know that there are obvious underlying state sub-spaces, SSD-RL is appropriate. However, when we have less understanding of our environment, other methods could be used more easily since it would be difficult to construct a state-space decomposition, and it would be hard to assess whether the decomposability factor of the state sub-spaces are appropriate for SSD-RL. In these cases, using PCA as described in [31] to speed up training could be a better option. PCA is an automatic process and does not require any prior knowledge about the environment. But with this method, we must also risk the loss of vital information during compression, which may lead us to converge to a sub-optimal policy.

Our method is also quite similar to Hierarchical Reinforcement Learning (HRL) methods we discussed in chapter 3. In fact, we can consider SSD-RL as a HRL method as we do split up a large problem into a hierarchy of sub-problems. We previously discussed Sutton et al.'s work which sped up training through hand-crafting sub-goals and splitting up the learning [22]. In comparison, our method does not involve manual sub-goal formulation, but rather manual state space decomposition. This can be more simplistic as we only need to know the state-space and some information about the environment's system dynamics. Whereas with sub-goal formulation, we not only need to know the state-space and system dynamics, but we also require knowledge of what the final policy is like so that we can come up with smaller policies which will help the agent progress to that. However, we do recognize that there are exceptions where finding sub-goals is more simple. Overall, both HRL and SSD-RL are shown to be effective, however, the nature of our problem greatly affects the performance of these two methods. Some problems have an obvious state-space decomposition, other problems could have clear sub-goals in the optimal policy which makes HRL easier to apply than SSD-RL. Therefore, SSD-RL and HRL both offer solutions to different groups of problems.

9.4 Summary

There are clearly many factors which we need to consider before using SSD-RL to speed up training. The type of Reinforcement Learning environment is arguably the most important. We need to be able to attain a suitable state-space decomposition so that SSD-RL can exploit that during training, and hence reduce training time. Crafting the state-space decomposition could be simple if we have adequate knowledge of our environment's system dynamics, but could also be time consuming if we don't. Therefore, it is vital to assess the nature of our problem before proceeding with SSD-RL. Other factors such as hyper-parameter tuning are relatively straight-forward in comparison. We have shown that the hyper-parameters are not highly sensitive to the performance of our method, and we have also proposed recommended values for them.

Chapter 10

Extending SSD-RL

In this chapter, we discuss how SSD-RL can be extended into a distributed method or a multi-agent method. We also highlight future work that can be done to extend our research.

10.1 Distributed SSD-RL

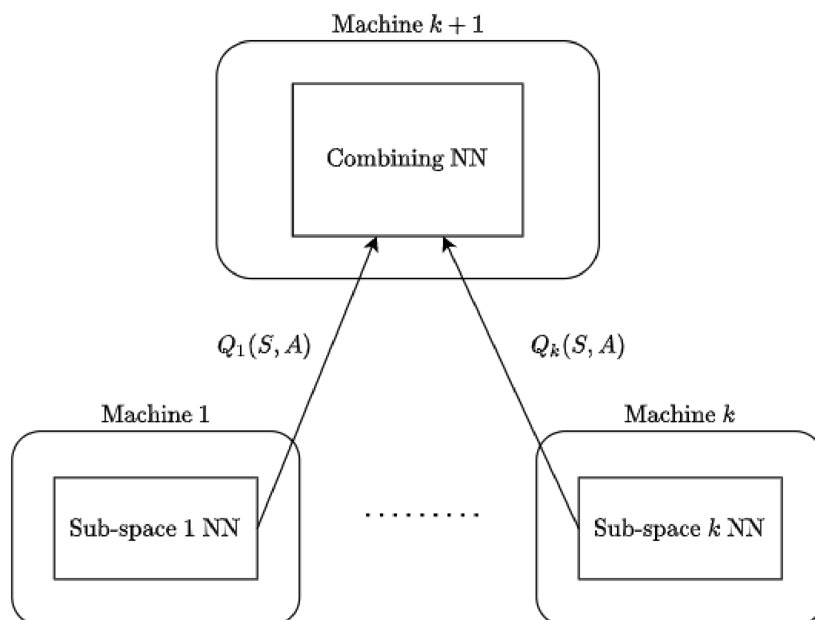

Figure 10.1: Illustration of SSD-RL as a distributed Reinforcement Learning algorithm

Reinforcement Learning usually requires access to a large amount of data for training. However, in a distributed system, data is rarely kept at a centralised location. Rather, we usually have data stored locally at different nodes. Therefore, transporting and aggregating all the data from each node to a central entity is limited by bandwidth and privacy leakage concerns. To address this problem, we need distributed algorithms which can work on multiple local data sets. SSD-RL was designed to tackle this issue as we can extend it into a distributed Reinforcement Learning method. In stage 1 of training, our sub-space neural networks are trained separately, so the computation for each of these can be carried out in separate machines (see figure 10.1). There is no need for global communication throughout state 1 of training. In stage 2 of training, SSD-RL only requires the global transfer of learned information, such as each sub-space network's weights or direct outputs. This significantly reduces the strain on communication bandwidth and ensures that no sensitive data risks being leaked in communication channels.

In fact, we are also able to use this extended method in non-distributed settings. Even though we aren't required to operate on distributed data sets, we could use the method to parallelise stage 1 of training and speed up learning even more.

10.2 Multi-agent SSD-RL

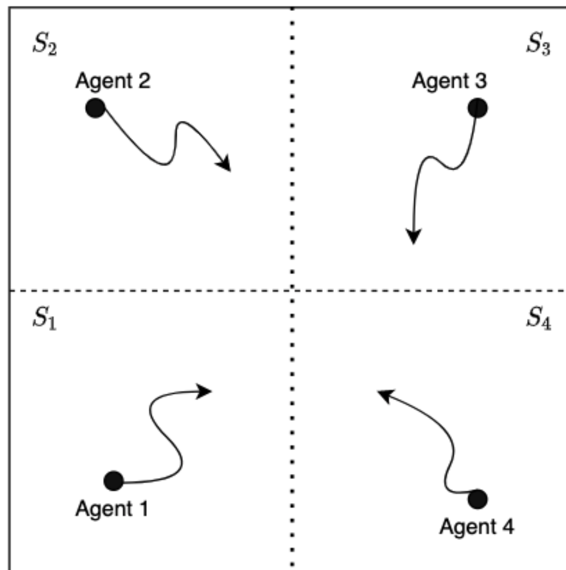

Figure 10.2: Exploration visualisation for Multi-agent SSD-RL. S_k denotes the kth state sub-space

Currently, SSD-RL only involves one exploring agent. Since we have multiple sub-space networks which are trained separately, we could explore the possibility of applying multi-agent reinforcement learning to our method. We could assign an exploring agent to each state sub-space, and each of the agents could train their respective sub-space network. Finally, we can devise an effective knowledge combining method. One way to do this is to create a separate entity which is responsible for collecting the inter-subspace experiences from the other exploring agents and training the combining network with them. Although, more investigation needs to go into how much the additional communication costs affect our performance.

The most intuitive way to use this method is to ensure that each agent starts exploring in their respective state sub-space. However, this may be difficult for some problems. For example, in our room maze environments, this is infeasible as starting in another room (or another state sub-space) is cheating the maze task as we aren't starting in the designated start position. On the other hand, this method could work in other problems where multiple agents are able to start in different states. For example, our method could work for feature detection in medical images. We could divide the state-space by splitting the image up into multiple sub-sections. Then, we can assign an agent to each of the sub-sections of the image to explore and try to detect features. This is possible as we know the dimensions of the image beforehand and there is no designated starting state we must be in for feature detection (see figure 10.2).

10.3 Future work

Investigating effects of the number of state sub-spaces

In our report, we have only experimented with splitting a state-space into two state sub-spaces. More investigation should go into how the number of state sub-spaces affects the performance of SSD-RL.

Evaluating SSD-RL against other state-of-the-art methods

In our report, we only quantitavely analysed SSD-RL's performance with Deep Q-Learning. More quantitative work should be done in seeing how SSD-RL performs against methods like Hierarchical Reinforcement Learning and Multi-agent Reinforcement Learning.

Develop and test distributed/multi-agent mode for SSD-RL

We discussed how SSD-RL could be extended into a distributed or multi-agent method, however, these extensions are not developed in our research yet. Theoretically, these functions are able to give us even more performance gain, but we need to develop it and actually test how effective it is for real-life problems.

Chapter 11

Conclusion

11.1 Summary

In our research, we successfully developed State-space Decomposition Reinforcement Learning (SSD-RL), a new Deep Reinforcement Learning (DRL) method which accelerates training by decomposing an environment's state-space into multiple state sub-spaces. We applied our method to several environments, including one based off a real-life dataset (Alibaba's cluster trace). In our evaluation against another state-of-the-art DRL method, we found that SSD-RL converged to the optimal solution up to 60% faster in grid-world environments, and seven times faster in our environment involving Alibaba's dataset. In fact, we found that the larger the state-space, the more performance gain we achieved, proving that SSD-RL can indeed help alleviate the effects of the state-space explosion problem. Not only did SSD-RL speed up convergence, but it also exhibited particularly fast and stable learning during the start of training. The ability to adapt quickly benefits Reinforcement Learning agents which need to learn within a continuously changing environment, and do not necessarily require convergence for a fixed environment.

Beyond designing and implementing SSD-RL, we gave a thorough evaluation about which types of environments would benefit most from our method. We demonstrated how leveraging the environment's system dynamics enables us to extract a more suitable state-space decomposition which allows SSD-RL to reduce training time even more. We also formulated a way to approximate how much performance gain we can achieve for an environment beforehand with the 'decomposability factor'.

Finally, we elaborated on how we can modify SSD-RL into a distributed method to address the challenges of running DRL in a distributed system. As many large real-life environments operate in a distributed manner, this particular extension of SSD-RL advances the practicality of applying DRL to a whole new realm of important applications.

Overall, our novel SSD-RL approach outperforms the state-of-art in multiple environments and addresses several limitations of using DRL in real-life settings. We are excited to see our approach develop further and be used in more applications.

11.2 Ethical discussion

As this project is an investigation into a general method for Deep Reinforcement Learning, there is no specific real-life application attached to the research. However, we utilise multiple environments to develop and evaluate SSD-RL. Our environments do not use any sensitive user data. With our grid-world environments, our data is entirely computer generated. With Alibaba's trace data set, the collected data is real, but similarly does not contain any user information or company specific information. The data set only contains statistics about machine metrics. All of the data sets we used are free and open to the public, and hence does not require any special permissions to use. We developed the grid-world environments by adapting a free open-source library called 'Minigrid' [34], and the Alibaba data set we used can be accessed publicly at https://github.com/alibaba/clusterdata/.

However, if we chose to use this method in other real-life applications, it is vital to consider the nature of the data we are using to perform Reinforcement Learning. If our environments include sensitive user data, we will need to obtain the users' permission to use it. We also need to make

sure that our Reinforcement Learning agent is secure and resilient against attacks which try to leak private data from its training and operation.

Next, must must also consider the risk of misuse. Reinforcement Learning algorithms discussed in our research are becoming more powerful and capable of solving extremely difficult problems. As there are a vast amount of problems that can be solved with Reinforcement Learning, there is always a possibility that developers use this method to create dangerous applications. For example, military applications or malicious agents which learn how to attack a system.

Looking at the bigger picture, machine learning training requires a significant amount of computation power, and therefore can consume a lot of energy. Although research is pushing for more efficient ways to train models, we must be mindful when running large machine learning experiments.

Appendix A

Hyper-parameters used in experiments

A.1 Baseline method: Deep Q-Learning

Q-network
Linear layer 1 (80 features)
ReLU activation
Linear layer 2 (80 features)
ReLU activation
Output layer

Hyper-parameters

- Learning rate: 0.0001 or 0.0002
- Batch size: 256

A.2 Our new method: State-space decomposition reinforcement learning

Sub-space network	Combining network
Linear layer 1 (50 features)	Linear layer 1 (35 features)
ReLU activation	ReLU activation
Linear layer 2 (50 features)	Linear layer 1 (35 features)
ReLU activation	ReLU activation
Output layer	Output layer with residual connection

Hyper-parameters

- N: 100-150 episodes
- T: 20 episodes
- Learning rate: 0.0001 for sub-space networks and 0.0002 for combining network
- Batch size: 256

Bibliography

1. Mwiti D. 10 Real-Life Applications of Reinforcement Learning. en-US. 2020 Jul. Available from: https://neptune.ai/blog/reinforcement-learning-applications [Accessed on: 2021 Jan 18]
2. garychl. Applications of Reinforcement Learning in Real World. en. 2018 Aug. Available from: https://towardsdatascience.com/applications-of-reinforcement-learning-in-real-world-1a94955bcd12 [Accessed on: 2021 Jun 9]
3. Kaelbling LP, Littman ML, and Moore AW. Reinforcement Learning: A Survey. arXiv:cs/9605103 1996 Apr. arXiv: cs/9605103. Available from: http://arxiv.org/abs/cs/9605103 [Accessed on: 2021 Jan 9]
4. Zhang Z, Ma L, Leung KK, Tassiulas L, and Tucker J. Q-Placement: Reinforcement-Learning-Based Service Placement in Software-Defined Networks. *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. ISSN: 2575-8411. 2018 Jul :1527–32. DOI: 10.1109/ICDCS.2018.00159. Available from: https://ieeexplore.ieee.org/document/8416422
5. Lazovskiy V. What's In Your Customer's Next Shopping Cart? en. 2018 Mar. Available from: https://towardsdatascience.com/whats-in-your-customer-s-next-shopping-cart-73d64287ec53 [Accessed on: 2021 Jan 19]
6. Abdi A. Three types of Machine Learning Algorithms. 2016 Nov. DOI: 10.13140/RG.2.2.26209.10088
7. Nagpal A. Clustering — Unsupervised Learning. en. 2017 Nov. Available from: https://towardsdatascience.com/clustering-unsupervised-learning-788b215b074b [Accessed on: 2021 Jan 18]
8. Srivastava S. State of Deep Reinforcement Learning: Inferring Future Outlook. en-US. 2020 May. Available from: https://www.analyticsinsight.net/state-deep-reinforcement-learning-inferring-future-outlook/ [Accessed on: 2021 May 30]
9. Zhang A, Lipton Z, Li M, and Smola A. Dive into Deep Learning — Dive into Deep Learning 0.16.0 documentation. Available from: https://d2l.ai/index.html [Accessed on: 2021 Jan 18]
10. Deisenroth MP, Faisal AA, and Ong CS. Mathematics for Machine Learning. en
11. Mnih V, Kavukcuoglu K, Silver D, Graves A, Antonoglou I, Wierstra D, and Riedmiller M. Playing Atari with Deep Reinforcement Learning. en :9. Available from: https://arxiv.org/pdf/1312.5602v1.pdf
12. Ashraf M. Reinforcement Learning Demystified: Markov Decision Processes (Part 1). Available from: https://towardsdatascience.com/reinforcement-learning-demystified-markov-decision-processes-part-1-bf00dda41690 [Accessed on: 2021 Jan 17]
13. Sutton RS and Barto AG. Reinforcement Learning: An Introduction. en
14. Martin M. Searching for optimal policies I: Bellman equations and optimal policies. en. 2011
15. Pashenkova E, Rish I, and Dechter R. Value iteration and policy iteration algorithms for Markov decision problem. 1997 Jan. Available from: https://www.researchgate.net/publication/2605845_Value_iteration_and_policy_iteration_algorithms_for_Markov_decision_problem

16. OpenAI. Part 2: Kinds of RL Algorithms — Spinning Up documentation. Available from: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html [Accessed on: 2021 Jan 18]
17. Maroti A. RBED: Reward Based Epsilon Decay. arXiv:1910.13701 [cs] 2019 Oct. arXiv: 1910.13701. Available from: http://arxiv.org/abs/1910.13701 [Accessed on: 2021 Jan 11]
18. Akanmu S, Garg R, and Gilal A. Towards an Improved Strategy for Solving Multi- Armed Bandit Problem. International Journal of Innovative Technology and Exploring Engineering 2019 Oct; 10:5060–4. DOI: 10.35940/ijitee.L2522.1081219. Available from: researchgate.net/publication/336650162_Towards_an_Improved_Strategy_for_Solving_Multi-_Armed_Bandit_Problem
19. Yang Y. Agile and Intelligent Locomotion via Deep Reinforcement Learning. en. Google AI Blog. Available from: http://ai.googleblog.com/2020/05/agile-and-intelligent-locomotion-via.html [Accessed on: 2021 Jan 8]
20. Sutton R, Mcallester D, Singh S, and Mansour Y. Policy Gradient Methods for Reinforcement Learning with Function Approximation. Adv. Neural Inf. Process. Syst 2000 Feb; 12. Available from: https://www.researchgate.net/publication/2503757_Policy_Gradient_Methods_for_Reinforcement_Learning_with_Function_Approximation
21. Jain D, Iscen A, and Caluwaerts K. Hierarchical Reinforcement Learning for Quadruped Locomotion. arXiv:1905.08926 [cs] 2019 May. arXiv: 1905.08926. Available from: http://arxiv.org/abs/1905.08926 [Accessed on: 2021 Jan 11]
22. Sutton RS, Precup D, and Singh S. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. en. 1998. Available from: https://scholarworks.umass.edu/cgi/viewcontent.cgi?article=1212&context=cs_faculty_pubs [Accessed on: 2021 Jan 11]
23. Sahni H, Kumar S, Tejani F, Schroecker Y, and Isbell C. State Space Decomposition and Subgoal Creation for Transfer in Deep Reinforcement Learning. arXiv:1705.08997 [cs, stat] 2017 May. arXiv: 1705.08997. Available from: http://arxiv.org/abs/1705.08997 [Accessed on: 2021 Jan 11]
24. Nachum O, Gu S, Lee H, and Levine S. Data-Efficient Hierarchical Reinforcement Learning. arXiv:1805.08296 [cs, stat] 2018 Oct. arXiv: 1805.08296. Available from: http://arxiv.org/abs/1805.08296 [Accessed on: 2021 Jan 11]
25. Jong NK and Stone P. Towards Learning to Ignore Irrelevant State Variables. en :6. Available from: https://www.researchgate.net/publication/2891229_Learning_to_Identify_Irrelevant_State_Variables
26. Tan M. Multi-Agent Reinforcement Learning: Independent vs Cooperative Agents. en :8. Available from: https://web.media.mit.edu/~cynthiab/Readings/tan-MAS-reinfLearn.pdf
27. Leroy G, Rueckert D, and Alansary A. Communicative Reinforcement Learning Agents for Landmark Detection in Brain Images. arXiv:2008.08055 [cs] 2020 Sep. arXiv: 2008.08055. Available from: http://arxiv.org/abs/2008.08055 [Accessed on: 2021 Jan 10]
28. Dean TL, Givan R, and Leach S. Model Reduction Techniques for Computing Approximately Optimal Solutions for Markov Decision Processes. arXiv:1302.1533 [cs] 2013 Feb. arXiv: 1302.1533. Available from: http://arxiv.org/abs/1302.1533 [Accessed on: 2021 Jan 20]
29. Ravindran B and Barto AG. SMDP Homomorphisms: An Algebraic Approach to Abstraction in Semi-Markov Decision Processes. en :6
30. Ferns N, Castro PS, Precup D, and Panangaden P. Methods for computing state similarity in Markov Decision Processes. arXiv:1206.6836 [cs] 2012 Jun. arXiv: 1206.6836. Available from: http://arxiv.org/abs/1206.6836 [Accessed on: 2021 Jan 20]
31. Curran W, Brys T, Taylor M, and Smart W. Using PCA to Efficiently Represent State Spaces. Available from: https://www.researchgate.net/publication/275897226_Using_PCA_to_Efficiently_Represent_State_Spaces [Accessed on: 2021 Jan 9]
32. Levy A, Konidaris G, and Saenko K. Learning multi-level hierarchies with hindsight. en. 2019 :16. Available from: https://openreview.net/pdf?id=ryzECoAcY7

33. Leung K, Zhang Z, Pritz P, Ma L, Bertino E, and Poularakis K. State Decomposition, Distributed annd Hierarchical Reinforcement Learning for SDC

34. Chevalier-Boisvert M, Lucas W, and Suman P. gym minigrid. 2018. Available from: https://github.com/maximecb/gym-minigrid

35. Ding H. Alibaba Cluster Trace Program. original-date: 2017-09-05T03:16:34Z. 2021 May. Available from: https://github.com/alibaba/clusterdata [Accessed on: 2021 May 17]