Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2021

Project Title:	**State-decomposition techniques for sample efficient Deep Reinforcement Learning**
Student:	**Edoardo David Santi**
CID:	**01373388**
Course:	**MEng Electrical & Electronic Engineering**
Project Supervisor:	**Prof. Kin K. Leung**
Second Marker:	**Dr Tania Stathaki**

Acknowledgments

I would like to thank my supervisor Prof. Kin K. Leung for the continuous support and availability throughour the duration of this project and my excitement about the prospect of working with him as my PhD supervisor in the future. I would also like to express my gratitude towards my parents for their continuous support and encouragement throughout my studies.

Abstract

Your abstract

Contents

Chapter 1

Introduction

1.1 Motivation

Reinforcement learning (RL) has been a very active research area in the last few years, achieving impressive feats in a wide range of applications. Some advantages of reinforcement learning algorithms are great flexibility, the capability of solving both very low and high-level problems [26] and not requiring a model of the environment in which they are used. RL has famously been used to solve games, such DeepMind's AlphaGo [24], a program that in 2016 beat the top-ranked Go player in the world. Other successful applications are in healthcare [34], robotics [2], autonomous driving [7] and natural language processing [21]. My project supervisor Prof. K. K. Leung has also been working on applications of RL in software-defined networks (SDNs), such as for the control of the placement of services [35] and the synchronisation of controllers in distributed SDNs [36].

Dealing with very large systems is one of the main problems encountered in the application of reinforcement learning in practical environments. RL is based on learning through interaction with the environment. As the size of the system increases, so does the complexity of the problem and it becomes increasingly difficult to learn how to interact with the environment, making the learning process longer. In practical scenarios, it might not be feasible to train an RL agent for long enough for it to be a feasible solution, as this might incur excessive costs or, in the case that the behaviour of the environment changes over time, the algorithm might not be able to learn fast enough to adapt to the changing environment.

1.2 Project Definition

The goal of this project is to develop a state-decomposition method, that aims to alleviate the problem of large state spaces of the Markov Decision Processes (MDPs) that underlay many practical environments. This new method leverages the characteristic of many control problems of being composed of many almost independent sub-problems that only seldom interact with each other. The method considers these sub-problems independently as well as combining them to form a global agent that controls the whole system and takes into account their interactions.

This project deals with developing reinforcement learning algorithms that use the high-level idea of state-decomposition and to test them and benchmark them against baseline algorithms to determine if such methodology can provide any performance gains. The performance is measured in terms of training samples or training episodes, that is the amount of training experience required to achieve a certain level of accuracy by the algorithm. This is not to be confused with reducing training time, which is more involved with reducing computational effort, which was not considered in this project.

1.3 Structure of the report

This project has the following structure. Chapter-2 provides the background information required to understand the topic of this project. This is about reinforcement learning basics and Q-learning and Deep Q-learning, as well as introducing the problem of large state-spaces which this project tries to alleviate and methodologies used in the literature for the same scope. Chapter-3 introduces the state-decomposition method more specifically, explaining the high-level ideas and the originally devised method to decompose state-spaces. Chapter-4 presents the tools and methods that were used in testing the state-decomposition technique and the type of experiments that were run. Chapter-5 shows the most important results obtained during testing and their interpretation. Chapter-6 discusses the applicability of the proposed techniques and provides a real-life example where the technique could be useful. Lastly, Chapter-7 discusses the findings of this project, its limitation and possible extension and future work.

Chapter 2

Background

2.1 Reinforcement learning

"The reinforcement learning problem is meant to be a straightforward framing of the problem of learning from interaction to achieve a goal" [26].

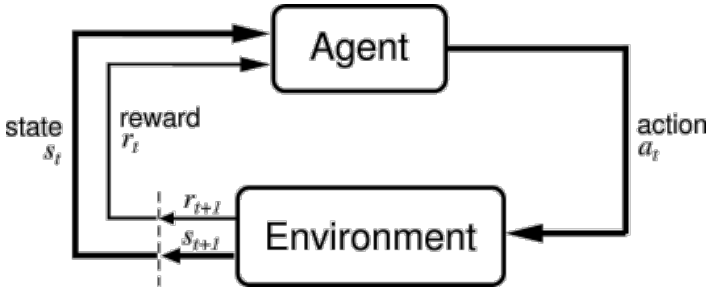


Figure 2.1: Interaction of the agent with the environment during time step t. Figure taken from [26]

The reinforcement learning *agent* continuously interacts with the *environment*. This interaction can be modelled as a sequence of time steps $t = 0, 1, 2, 3, ...$[1]. At each step t, the environment is represented by its *state* $S_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of all the possible states. Taking the *state* as input, the agent outputs an *action* $a_t \in \mathcal{A}$, where $\mathcal{A}$ is the set of all the possible actions. In the next time step $t+1$, the environment transitions to a new *state*[2] S_{t+1} and emits a *reward* $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, which the agent uses as feedback information to "learn" how to interact with the environment. Each time step can then be represented by the $\langle S_t, A_t, R_{t+1}, S_{t+1} \rangle$ tuple.

The agent implements a mapping from the possible states to the probabilities of choosing the different actions. This mapping is called *policy* and at time step t, it adopts symbol π_t. The probability of choosing action $A_t = a$ given the current state $S_t = s$ is denoted by $\pi_t(a|s)$. The goal of reinforcement learning is to learn the optimal policy, which is the policy that maximises the rewards, through repeated interaction with the environment. In general, we aim to maximise a function of the rewards at all time steps rather than the immediate reward, so the optimal policy doesn't necessarily maximise immediate rewards. The simplest case is to maximise

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{2.1}$$

where T is a final time step. This approach can be used when the agent-environment interaction has an end, in which case a sequence of interactions from the initial time to time T can be called

[1] The interaction with the environment can also be modelled in continuous time, however this is outside the scope of this project.

[2] Note that it is possible that $S_{t+1} = S_t$

an *episode*. In other cases, the interactions can continue indefinitely, thus this reward function is not feasible as $T = \infty$ and G_t could be infinite. To fix this, we apply a *discount rate* γ, with $|\gamma| < 1$, to future rewards:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.2}$$

This allows G_t to have a finite value in continuing tasks, which are those tasks that don't have finite time episodes.

In general, the dynamics of the environment could depend on the whole history of states, rewards and actions over an episode. However, we aim to have an agent which can choose the optimal action at each time step based only on the current state. Thus the knowledge of the current state and action should give the maximum possible amount of information about the dynamics of the environment. This is true if knowledge of the history of states, rewards and actions in the previous time steps does not provide any additional information about the dynamics of the environment than knowing the current state. States that follow this property are said to follow the *Markov property*. More formally, if the states follow this property

$$Pr\{R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t\} \tag{2.3}$$
$$= Pr\{R_{t+1} = r, S_{t+1} = s' | S_t, A_t\} \tag{2.4}$$
$$= p(s', r | s, a) \tag{2.5}$$

This is the equation that governs Markov Decision Processes (MDPs) and hence a reinforcement learning task formulated with states that follow the Markov property is an MDP.

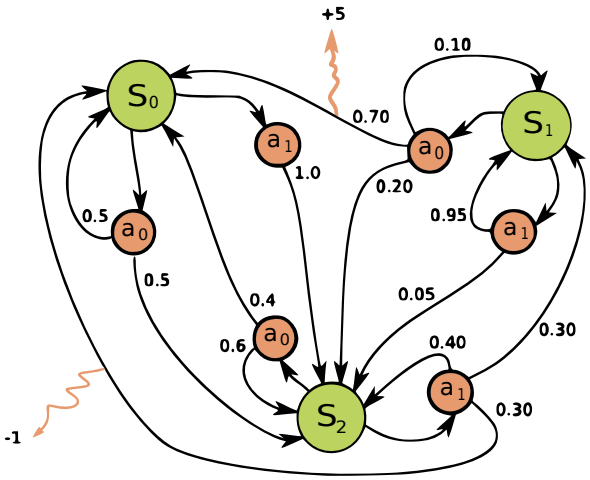


Figure 2.2: Graph representation of a Markov Decision Process. Note the rewards (zigzag arrows) that are released in certain transitions. [3]

Most reinforcement learning algorithms involve the estimation of the *value functions*, which are functions of states or state-action pairs. A value function represents how good it is to be in a certain state or to perform a certain action in a certain state, in terms of the expected return G_t and given that we are following a certain policy. The action-value function is defined as

$$q_\pi(s, a) = \mathrm{E}_\pi[G_t | S_t = s, A_t = a] = \mathrm{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right] \tag{2.6}$$

Similarly, the state-value function is given by

$$v_\pi(s) = \mathrm{E}_\pi[G_t | S_t = s] \tag{2.7}$$

$$= \mathrm{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right] \tag{2.8}$$

$$= \sum_a \pi(a|s) \, \mathrm{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right] \tag{2.9}$$

$$= \sum_a \pi(a|s) q_\pi(s,a) \tag{2.10}$$

It is useful to specify the value functions in recursive format, known as the Bellman equations. This gives:

$$v_\pi(s) = \mathrm{E}_\pi[G_t | S_t = s] \tag{2.11}$$

$$= \mathrm{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \tag{2.12}$$

$$= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + v_\pi(s')] \tag{2.13}$$

and

$$q_\pi(s,a) = \mathrm{E}_\pi[G_t | S_t = s, A_t = a] \tag{2.14}$$

$$= \mathrm{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \tag{2.15}$$

$$= \sum_{s',r} p(s', r|s, a)[r + v_\pi(s')] \tag{2.16}$$

For finite MDPs it can be proved that there always exists an *optimal policy* that maximises the state-values of all states the action-values of all state-action pairs, giving the optimal state-values $v_*(s)$ and $q_*(s,a)$:

$$v_*(s) = \max_\pi v_\pi(s), \quad \forall s \in \mathcal{S} \tag{2.17}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a), \quad \forall s \in \mathcal{S}, \quad \forall a \in \mathcal{A} \tag{2.18}$$

Given full knowledge of the dynamics of the system[3] and a fixed policy, we are able to evaluate the optimal state-values and action-values using methods based on dynamic programming [9]. One such method called *iterative policy evaluation*, which adopts the update step based on the Bellman equations given below, can be shown to converge to the state-values:

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left[R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s, s', a) v_k(s') \right] \tag{2.19}$$

Policy iteration is one possible method of evaluating the state values when the optimal policy is not given, which alternates steps of policy evaluation and updates of the policy. The policy is updated so that in every state the greedy action, meaning the one with the highest action-value, is always picked. This process of solving the MDP when given the model of the environment is known as *planning*.

However, in many applications the model of the environment is unknown. Given a certain policy, it is possible in this case to approximate the state and action values by sampling the interaction with the environment. Monte Carlo methods interact with the environment by following the given policy and approximate each state-value using the returns obtained in each episode after the first or each visit to the state. The update that occurs every time the state occurs for the first time in the episode (or every time, depending on the version of the algorithm) is

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{2.20}$$

[3] The probabilities of state-transitions and rewards given the current state-action pair

The disadvantage is that state-values cannot be updated until the end of an episode. Temporal Differences methods are another class of methods that solve this problem. Instead of relying on returns (G_t), they approximate them as the sum of the reward and the discounted state-value of the next visited state. The simplest TD algorithm's update step is given by

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{2.21}$$

There are many applications of reinforcement learning in which state-space and action-spaces are continuous. In that case, the MDP is not finite as there is an infinite number of possible states and actions. This will not be discussed as it is outside of the scope of this project, which only deals with discrete state and action spaces.

2.2 Neural networks

Should I give a bried intrdoction to neural networks

2.3 Q-learning and Deep Q-learning

SHOULD ADD A BIT OF INFORMATION ABOUT NEURAL NETWORKS

Many RL algorithms deal with the problem of solving an MDP without being given the optimal policy and the dynamics of the system. Q-learning [32] is one of the most noteworthy among these. It is based on Temporal Differences learning as the action-value updates follow a very similar formula. The basic value update in Q-learning is given by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, a)] \tag{2.22}$$

Differently from the previously described TD algorithm, this operates on action-values instead of state-values. In Q-learning, the value update is done using the term $\max_a Q(S_{t+1}, a)$, which is the action-value of the greedy-action in the next visited state. This is not necessarily the action taken in the following step when visiting S_{t+1}, thus this an *off-policy* algorithm.

Differently from the Temporal Differences algorithms that aim at evaluating a fixed policy, Q-learning also deals with the control problem of choosing the policy. Q-learning is normally programmed so that its current policy is based on its current action-value estimates. An example is the ϵ-greedy policy, in which at each time step, the greedy action is picked with probability $1 - \epsilon$ and a random action is picked with probability ϵ.

This algorithm is only defined for discrete environments and it is proven to converge to the optimal action-values with a probability of 1, as long as all the states and actions are visited multiple times [31]. This requirement might not be feasible when the number of states is very large and it is impossible when the number of states is infinite such as when the MDP's state-space is continuous. Using a learned Q-function approximator instead of simply storing the action-values in tabular form can solve this issue, as it is possible to learn a mapping function that generalises over the whole state-space without having to visit all the possible states.

Deep Q-Learning (DQN) [17] implements the Q-function approximator using a deep neural network. This was demonstrated to achieve state-of-the-art performance in learning to play Atari games directly from the pixel data [16]. DQN takes advantage of the ability of deep learning to learn from high-dimensional data representations to be able to directly feed large state-spaces into the agent. Deep neural networks rely on the assumption that they are fed independent samples, however in reinforcement learning the samples are correlated as they are generated by a temporal sequence. An

experience replay is used to solve this. The transition samples are saved in memory and a random batch is selected to train the neural network at each time-step, thus removing correlations due to temporal proximity of the samples. This also allows the update to be based on a batch, rather than a single sample, which improves the sample efficiency[4].

One issue of the Q-learning algorithm is that using the maximum action value of the next visited state in the update step results in a positive bias in the estimation of the action values, which can lead to poor performance in some stochastic environments. Double Q-learning [33] deals with this problem by splitting the samples randomly between two separate sets of action value estimates and by updating the values of each set of estimates using the values of the other, which can be proven to remove the positive bias[5]. The same problem persists in deep Q-learning and a similar solution has been devised in Double Deep Q-Learning [29]. Another version of DQN that achieved improved performance in typical benchmarks such as playing Atari games is Dueling DQN [30], which operates on the concept of expressing action values as a sum of state values and advantages: $\mathcal{Q}_\pi(s,a) = \mathcal{V}_\pi(s) + \mathcal{A}_\pi(s,a)$.

While the original Q-learning algorithm can only be applied to systems with discrete state and action spaces, other algorithms based on Q-learning have been proposed to deal with continuous state and action spaces, such as Wire Fitted Neural Network Q-learning [12].

2.4 Problem of large state-spaces

Very large systems correspond to MDPs with a very large number of states, which can be difficult to solve using reinforcement learning algorithms. The number of possible state-transitions is the square of the number of states, thus the complexity of controlling a system grows very quickly as the size of a system increases, and learning an optimal policy becomes very difficult and often infeasible.

Various approaches have been used to solve this issue. One general approach is to learn representations of the states to improve generalisation to unseen states or to decrease the dimensionality of the states by compression. DQN [16] exploits the ability of deep learning to learn from high-dimensional data and to generalise to solve this issue, though this is often not sufficient. [22] jointly learns state and action embeddings to improve generalisation. In World Models [13], a compressed latent representation of the states is explicity learned by an autoencoder, which is then fed into the reinforcement learning model instead of the original states. Other methods rely on state aggregation [10], in which states with similar state-transition probabilities and rewards are grouped to form a higher-level state. This technique can also be applied to hierarchical reinforcement learning to define the states of the global learner [6].

Hierarchical reinforcement learning [8] is another approach that reduces the impact of large state spaces. Hierarchical RL provides temporal abstraction by subdividing the main task into sub-tasks. For example, the task of commuting to work can be subdivided into sub-tasks such as opening a door, turning right, walking to the tube station, etc. The agent is composed of hierarchies of sub-agents in which each sub-agent is fed a compressed or reduced state space in input and outputs high-level actions to its ”children” sub-agents that perform the lower-level actions required to complete the received higher-level action. Examples of hierarchical reinforcement learning algorithms are based on MAXQ value decomposition [11], Options [27] and Hierarchies of Machines (HAM) [20].

Model-based reinforcement learning [18] has been receiving increasing attention lately. This class of algorithms is at the intersubsection of planning algorithms and model-free reinforcement learning as it approximates the model of the environment during the learning process, which can be used to

[4] How efficiently the samples are used by the algorithm to learn. A more sample efficient algorithm requires fewer interactions with the environment to learn an optimal policy.

[5] It actually introduces a small negative bias, thus tending to underestimate the action values.

more efficiently learn from new samples. Dyna [25] learns a model of the environment and uses it to simulate samples for Q-learning other than those obtained by actual interaction with the environment. AlphaZero [24], which achieved state-of-the-art performance in playing Chess, leverages full knowledge of the model[6] to perform a Monte-Carlo tree search for every step in the real-world environment. [13] trains an RNN model that predicts the following state at each-time step. Though this class of algorithms does not explicitly tackle the problem of large state-spaces, the improvements in sample efficiency can still solve the issue by allowing to train a model with fewer interactions with the environment.

[6] The model is given at the start rather than learnt.

Chapter 3

State-Decomposition method

The main topic of this project is the development of the state-decomposition method. This is suited to control problems of environments that are composed of many sub-problems that only seldom interact with each other. The state-decomposition method takes advantage of the almost independence of these problems to learn agents that deal with these smaller problems, which are easier to solve. These separate agents need to be combined to take into account the interactions between the sub-problems. An example of a system composed of many almost independent sub-systems could be the control of a communication network composed of many networks that rarely communicated with each other. The state-decomposition method would aim to learn an optimal controller by learning agents that control the smaller sub-networks and combining them to account for the small amount of interaction between sub-networks.

3.1 The original state-decomposition method

The first step of the process is to separate the sub-problems by identifying separate groups of states. This is done by examining the state-transition probability matrix. Given a finite MDP, this is defined as the matrix where each entry P_{ij} is the probability of transitioning from state $S_t = i$ to the next state $S_{t+1} = j$:

$$\begin{bmatrix} P(S_{t+1}=s_0|S_t=s_0) & P(S_{t+1}=s_1|S_t=s_0) & ... & P(S_{t+1}=s_n|S_t=s_0) \\ P(S_{t+1}=s_0|S_t=s_1) & P(S_{t+1}=s_1|S_t=s_1) & ... & P(S_{t+1}=s_n|S_t=s_1) \\ ... & ... & ... & ... \\ P(S_{t+1}=s_0|S_t=s_n) & P(S_{t+1}=s_1|S_t=s_n) & ... & P(S_{t+1}=s_n|S_t=s_n) \end{bmatrix} \tag{3.1}$$

where n is the number of states in $\mathcal{S}$.

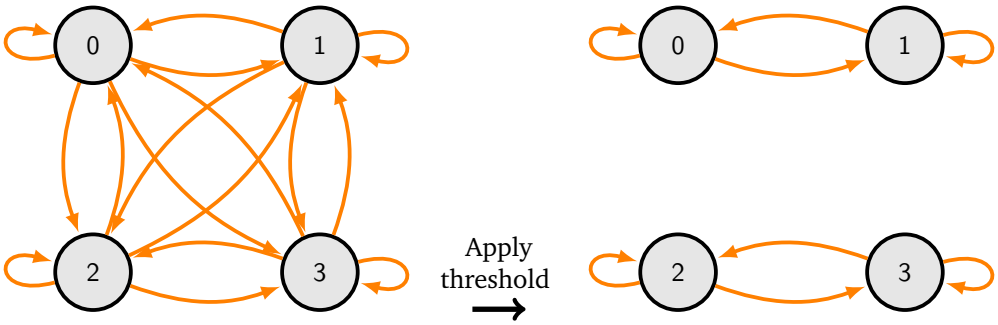


Figure 3.1: Applying a threshold to the state transition matrix and normalising the probabilities can cause the MDP to separate into separate MDPs as it reduces the number of possible transitions.

The state-decomposition method is based on the fact that some state-transition probabilities are negligible compared to others and could be regarded as 0 to obtain a sparse version of the state-

transition matrix. Specifically, every element smaller than a certain threshold τ can be set to 0. The probabilities of the matrix can then be normalised so that each row of the matrix has a sum of 1. This step is unnecessary and can be skipped in the implementation; however, it is necessary to obtain a valid MDP from the state-transition matrix after applying the threshold. Following this procedure, the original MDP could have transformed into multiple independent MDPs with smaller state-spaces, as shown in Figure-3.1. In this case, by reordering the states it is possible to express the state-transition matrix as a block matrix in the form:

$$\begin{bmatrix} \boxed{B_1} & 0 & ... & 0 \\ 0 & \boxed{B_2} & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & \boxed{B_n} \end{bmatrix} \tag{3.2}$$

where the matrices $B_1, B_2, ..., B_n$ are the state-transition matrices of the newly formed separate MDPs.
This is the method of state-decomposition that this report focuses on, which is based on ignoring state transitions that are below the threshold τ. There are different ways to decompose the state-space, the choice of which might depends on the specific problem that the method is applied to. A further discussion on this can be found in section-7.3.

The following describes the original architecture that was devised to apply state-decomposition to a practical problem. Once the states of the original MDP have been separated into multiple sub-spaces, it is possible to proceed with training the RL agent. In the first stage of training, a separate DQN agen is trained for each of these sub-spaces. To do so, we consider for each sub-space's agent only those state transitions that start and end within the same sub-space. Once the sub-spaces' agents converge, these should be combined to form a global agent that also takes into account transitions between the different sub-spaces. This is done by feeding the action values of the sub-spaces' agents, which are the outputs of their neural networks, as inputs of another neural network, as in Figure-3.2. This forms a bigger neural network which is the Q-function approximator of a new global DQN agent, which is then trained using all possible transitions. This will be referred to as the second stage of training.

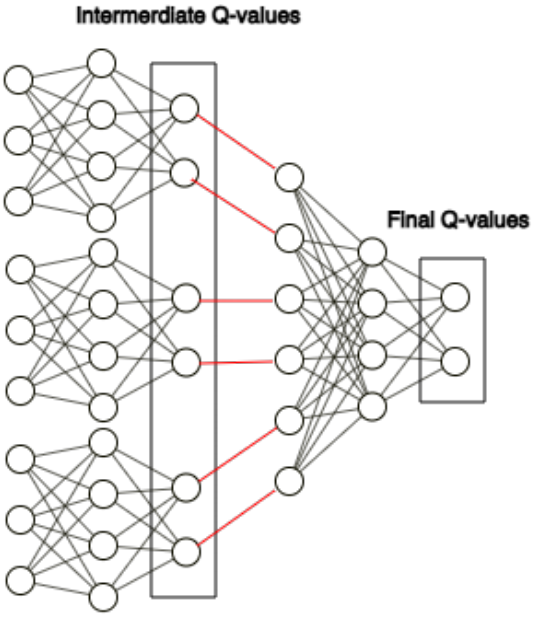


Figure 3.2: The NNs of the decomposed states' agents are merged as input of another NN to form a new bigger NN.

This method operates on the assumption that it would require fewer samples to achieve convergence for RL agents operating on many smaller MDPs than on one single MDP having as the number of states the sum of the number of states of the smaller MDPs. This is because the number of possible transitions in an MDP is the square of the number of states, so by dividing the system into separate independent sub-problems we reduce the complexity of the control problem[1]. By then combining the neural networks as previously explained, we take into account interactions between the sub-problems, allowing us to achieve the optimal policy for the original MDP. For this method to be successful, it is assumed that the weights of the neural networks of the sub-spaces' agents are not far from the values they take once the global agent converges in the final stage of training. This is equivalent to saying that the first stage of training provides a very good initialisation of the weights of the network used in the second stage. Although we use DQN agents, this method could be generalised to any agents that use a function approximator to evaluate the action-values given in input the states. This method however wouldn't be as useful if no approximator is being used, and the action-values are simply stored in a table, such as in the original Q-learning algorithm. In this case, applying the state-decomposition would simply ignore the transitions between different sub-spaces without changing the structure of the agent at all in the first stage of training.

It is important to mention that the state-transition matrix depends on the policy being followed by the agent that interacts with the environemnt. Thus the state-transition matrix is most likely different when starting to train an agent and upon convergence. For this reason, different state-decompositions could be possible depending on the policy that the state-transition matrix is based on. For example, some state-decompositions might exist regardless of the policy, such as when the transitions between two sub-spaces are made unlikely by the characteristics of the environemnt itself, such as moving between two place separated by a wall in a navigation problem. However, some decomposition are purely created by the policy of the agent, such as if in a navigation problem there is no wall seaparating two positions, but the agent simply never moves between these two places. More concrete examples of this are given in the next chapter and for now it is sufficient to acknowledge that different decompositions can occur depending on the assumed policy.

This method is the one that was initially proposed to study the effects of state-decomposition. In chapter-5 we show that this architecture is problematic, having issues such as the first stage of training not successfully speeding up the second stage.

3.2 Decomposing the state transition matrix

This report focuses on the application of state-decomposition where the state-decomposition is 'intuitive' and it is done in a non-mathematical way, as will be shown in the next sections. However this is not always the case, thus we need a systematic way to obtain the correct state-decomposition, given that we know the state-transition matrix. This would normally also be unknown, however, techniques to estimate this could be discussed as an extension of this project.
The decomposition into sub-spaces of states given the state-transition table is done using Algorithm-1.

Algorithm 1: State-decomposition algorithm.

stateTransMatrix: input 2D array representing the state-transition matrix
subspaces : output data structure representing the decomposed sub-spaces

```
subspaces = initialise as one subspace for each group;
for i ← 0 to height(stateTransMatrix)-1 do
    for j ← 0 to width(stateTransMatrix)-1 do
        if stateTransMatrix [i][j] > τ then joinSubspacesContaining(i, j);
    end
end
```

[1] By decreasing the number of possible transitions, as $\sum_{i=0}^{n} x_i^2 \leq (\sum_{i=0}^{n} x_i)^2$

Sometimes we know the number of sub-spaces that the original MDP should be decomposed into. For example, we might know the characteristics of the system and the number of subsystems it can be considered to have, such as how many sub-networks compose a certain communication network. In such case, the function that returns the decomposed sub-spaces can be run repeatedly while updating the threshold value τ at each iteration using binary search, until the MDP is decomposed in the correct number of sub-spaces. This is given in Algorithm-2.

Algorithm 2: State-decomposition into given number of sub-spaces

stateTransMatrix : input 2D array representing the state-transition matrix
nSubspacesDesired: input integer representing the number of sub-spaces to decompose into
subspaces : output data structure representing the decomposed sub-spaces
maxIterations $\leftarrow 50$;
changeFactor $\leftarrow 10$;
$\tau \leftarrow 1.0$;

upperBound $\leftarrow$ NONE;
lowerBound $\leftarrow$ NONE;

for $i \leftarrow 1$ **to** maxIterations **do**
  if upperBound *and* lowerBound *are defined* **then**
    $\tau \leftarrow$ (upperBound + lowerBound)$/2$;
  end
  if *only* upperBound *is defined* **then** $\tau \leftarrow$ upperBound$/$changeFactor;
  if *only* lowerBound *is defined* **then** $\tau \leftarrow$ lowerBound $*$ changeFactor;

  subspaces $\leftarrow$ `stateDecomposition` (stateTransMatrix, τ);

  if `nSubspaces`(subspaces) $==$ nSubspacesDesired $- 1$ **then** break;
  if `nSubspaces`(subspaces) $<$ nSubspacesDesired $- 1$ **then** lowerBound $= \tau$;
  if `nSubspaces`(subspaces) $>$ nSubspacesDesired $- 1$ **then** upperBound $= \tau$;
end

Chapter 4

Design and testing of the proposed decomposition technique

The state-decomposition method was implemented and tested in different environments and using different architectures. This section, describes the experimental method used to assess the performance of the state-decomposition method.

4.1 Tools

Python The programming language chosen for this project is Python. This is motivated by Python being a high-level language that allows writing compact, readable code and by the widespread availability of resources and packages for it. Additionally, Python can be run on Colab notebooks, which are described in the next paragraph.

Google Colaboratory (Colab) Colab is a platform from Google which is widely used for machine learning research and prototyping of new models. Colab combines a notebook platform that many Python developers are familiar with and the high-performance of cloud computing. The user can choose to run a program using CPUs, GPUs, or TPUs[1]. Examples of the available GPUs are Nvidia K80 and NVidia P100D. The disadvantage of Colab compared to cloud computing platforms such as Google Cloud and Amazon AWS is that the user is not guaranteed full access to the hardware accelerators (such as GPUs) and thus the performance can vary. The session length is also limited to 12 hours. These problems are mitigated by purchasing a ”Pro” license for the price of $10 per month, which prioritises access to GPUs and increases the maximum session length to 24 hours. The level of performance offered by Colab Pro is enough for this project. The main advantage is an easy to use platform thanks to its notebook format, rather than the less user-friendly access via SSH in the Terminal required by other cloud-computing platforms. This allows easy prototyping and makes it easy to make small changes to the code.

Keras The deep-learning models were implemented using the deep-learning API Keras [28]. This is a high-level interface that uses Tensorflow [5] for its backend. Keras allows to code deep-learning models in a very readable and compact way. Having Tensorflow as its backend, it can run on CPUs, GPUs, and TPUs, taking advantage of the high-performance hardware offered by Google Colab.

OpenAI Gym As this project deals with the development and evaluation of a new method for reinforcement learning algorithms, it was decided to apply it in a ”toy environment” that would simplify experimentation. OpenAI offers an open-source Python toolkit called ”OpenAI Gym” [19], which is currently the most commonly used toolkit to benchmark reinforcement learning algorithms. This

[1] Tensor Processing Unit: a custom hardware accelerator developed by Google to accelerate the linear algebra operations that occur in machine learning. [1]

toolkit offers a variety of different environments, ranging from classical control problems such as "CartPole-v0" [4], to more complicated environments such as playing Atari games using the pixel data as sensory input or solving robotic manipulation problems. The "OpenAI Gym" offers environments with a wide range of complexities and all combinations of discrete and continuous state and action spaces.

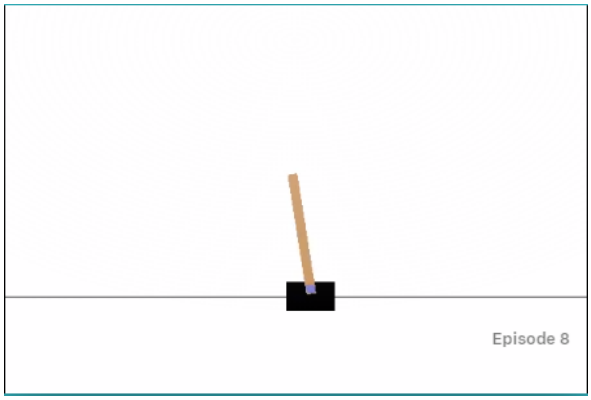


Figure 4.1: The CartPole-v1 environment is one of the most well-known environments of the OpenAI Gym library in which the goal is to balance the cartpole while keeping it within the boundaries of the screen by applying a leftwards or a rightwards force at each time-step. Screenshot of [4]

4.2 The need for a suitable environment

The state-transition matrices and their decomposed versions were obtained[2] for 4 different OpenAI Gym environments: "Cartpole-v1"[3], "FrozenLake8x8-v0", "CliffWalking-v0" and "Taxi-v3". The state-decomposition algorithm is aimed at environments whose state-spaces decompose into similarly sized sub-spaces, as this is the case in which the method would be potentially very effective. If one of the state-spaces is much greater than the others and similar in size to the original state-space, the reduction in complexity of the training in the first stage would not compensate for the more complex multi-stage nature of the state-decomposition method. Having sub-spaces that are too small (in the extreme case having just one state) limits the amount of learning that can be achieved in the first stage of training, simply delaying this to the second stage. The MDPs of "Cartpole-v1", "FrozenLake8x8-v0", "CliffWalking-v0", always decompose into one big sub-space and sub-spaces formed by only one state. This occurs because these environments are not formed of almost independent sub-problems that rarely interact with each other.

The "Taxi-v3" environment, shown on the left in Figure-4.2 was then deemed to be a more suitable candidate. The environment has:

- 500 discrete spaces, corresponding to 25 taxi positions in the 5x5 grid, 4 possible destinations, and 5 possible passenger positions (1 of which is inside the taxi).
- 6 discrete actions: 4 moves (up, down, left, right), pick-up passenger, and drop-off passenger.
- Rewards: +20 for a successful drop-off, -1 at each time step (also when hitting a wall), and -10 for illegal pick-up and drop-off actions.

An episode terminates when the passenger is successfully dropped off or after 200 time-steps. Applying the state-decomposition to this environment can form 4 equally sized sub-spaces even with a threshold of 0, meaning that these are 4 completely independent sub-spaces that can be treated as independent Markov Decision Processes. These 4 sub-spaces correspond to the 4 possible destinations

[2]These were obtained by "cheating" by simply inspecting the source code of the environments.
[3]This environment has continuous states which are discretised using a linear quantiser to apply the decomposition algorithm.

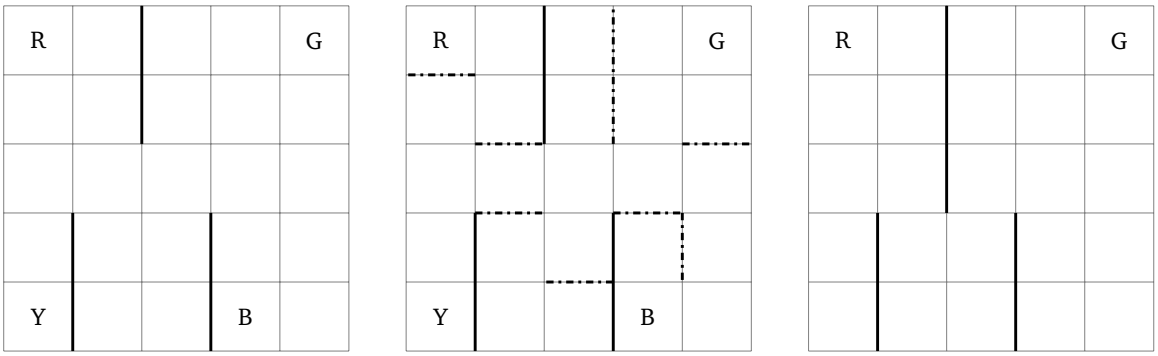
of the passenger, which never change during a single episode. Thus, this a more specific and easier problem than the one we are trying to solve, where the subsystems, though rarely, do interact with each other. Thus, this environment is not suitable for testing the state-decomposition algorithm as good performance of the algorithm in this simpler environment does not imply good performance in the more general scenario.

The "Taxi-v3" environment is composed of 4 independent MDPs. Modifying its transition-probabilities so that interaction between them is possible[4], would make this environment suitable for testing the decomposition algorithm. For this purpose, I proposed a modified version called "TaxiTraps", with added "traps" located on the edges of certain squares. These activate with a low probability p_{trap} when the Taxi moves over them. If a trap does not activate, the taxi simply moves over it as in the original "Taxi-v3" environment. When the trap does activate, the behaviour of the environment is modified in two ways:

1. The destination of the Taxi changes. This makes it possible for the destination of a passenger to change during an episode, thus providing the transitions between different sub-spaces that exist in the general problem that the state-decomposition method is aimed at.

2. A highly negative reward is released. This was added to ensure that the optimal policy in the "TaxiTraps" environment is different than in the original Taxi environment. Since in the first step of the state-decomposition method the transitions between different state sub-spaces are ignored, the agents are trained in an environment equivalent to the original "Taxi-v3". If the optimal policy is the same in "Taxi-v3" and "TaxiTraps", the optimal policy would not change between the first and second training stage of training, which is unlikely to happen in a general scenario. Thus, ensuring that the optimal policy changes between training stages is required for the results of testing to generalise to other environments.

During testing it was noted that such environments would take a very high number of iterations to train and would often get stuck for long periods of time at reward -200, which is when the agent has learned to not commit an illegal moves but it hasn't learned to drop-off the passenger successfully. Due to the limited computing power of my tools, this was a problem as it did not allow for significant testing with a high number of trials. Thus, I decided to try using a simpler version of the 'Taxi' environment. The number of destinationations was reduced to 2 and the multi-stage nature of the task, having to pick-up the passenger, navigate the grid and then drop it off was eliminated by setting the passenger to always be inside the Taxi, so that the pick-up and drop-off actions are eliminated.



(a) "Taxi" **(b)** "TaxiTraps" **(c)** "Taxi2"

Figure 4.2: Original and modified 'Taxi' environments.

While 'Taxi2' reduced the complexity of training, this was still not simple enough and it didn't allow for easy interpretability of the results. I then introduced the 'Grid' environment, which is a simple 25

[4]Making transition-probabilities between spaces of different subsystems non-zero.

by 25 grid with two destinations, as shown in 4.3. At the start of the episode, the taxi is initialised in a random position, while the passenger is always assumed to be inside the Taxi, as in 'Taxi2', so that the pick-up and drop-off actions are eliminated and the learning process only deals with learning how to navigate in the grid. The two destinations are located at opposite corners and the initial destination is always chosen to be the one closest to the initial position of the taxi. Throughout an episode the destination is always the same as in the initial state and the episode terminates when the taxi reaches the correct destination. To summarise, the environment has:

- 1250 discrete states, corresponding to 625 taxi positions in the 25x25 grid and 2 possible destinations.
- 4 discrete actions: move up, down, left and right.
- Rewards: +20 for being in the same position as the destination, -1 at each time step (also when hitting a boundary and thus not moving).

This is a perfectly decomposable environment, as there are no possible transitions between states having different destinations. For a complete evaluation of the state-decomposition method, it needs to be tested in a more generic environment, with non-zero albeit small transition probabilities between the decomposed sub-spaces. This was achieved by introducing a probability ϵ of changing the destination whenever the current destination is the one closest to the current position in the grid. The initial destination can now be the far one, with probability ϵ. This modified 'Grid' environment will be referred to as 'GridEps'. Table-4.1 summarises the transition probabilities in 'GridEps'.

		next state	
		destination 1	destination 2
state	triangle 1 - destination 1	$1-\epsilon$	ϵ
	triangle 1 - destination 2	0	1
	triangle 2 - destination 1	1	0
	triangle 2 - destination 2	ϵ	$1-\epsilon$

Table 4.1: State transition probabilities in the 'GridEps' environment.

I decided to set the transition probabilities this way as while providing small transition probabilities between different sub-spaces, it also modifies the policy as the value of ϵ changes. This is important as the case in which the optimal policy is unaffected by the value of ϵ is a special case which should not be used for testing.

Another version that was used in various test is very similar to 'GridEps' and will be caleed 'GridEps2'. The difference is that the probabilities with which the destination is picked is slightly different. In this case, when the destination is the close one, the destination in the next state is the close one for next state with probability $1-\epsilon$ and the far one for the next state with probability ϵ, whereas in in 'GridEps' the destination in the next state depends on the position of the previous state. This only makes a differnce in transitions that cross the boundarh between the two triangles. In 'GridEps', when the boundary is crossed and the destination was the close one, the destination most likely remains the same, while in 'GridEps2' it most likely switches to the other one, which is the closest after crossing the boundary. This may seem like a small difference, but it will be shown that is has a great impact on the effect of different state-decompositions.

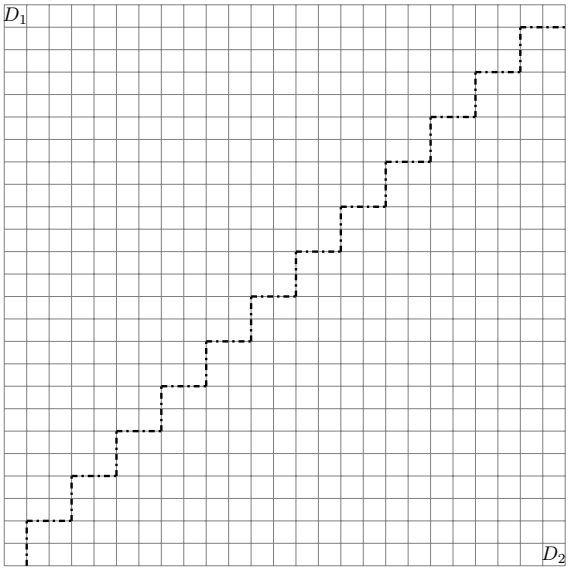


Figure 4.3: 'Grid' environment. The dotted lines have no effect on the environment, they simply separate what will be referred to as the upper-left triangle and the lower-right triangle.

This environemnt is an example of a scenario in which the decomposition is "intuitive", as mentioned in section-3.2. It is clear here that one possible decomposition is given by creating one sub-space for each possible destination, as at each-time the probability of changing destination is very small, being either 0 or ϵ. Another kind of decomposition that was tested, altough it doesn't necessarily follow the procedure previously described for decompositing the state-transition matrix for $\epsilon > 0$, is to assign states corresponding to positions in the upper-left "triangle" of the grid to one sub-space and positions in other other "triangle" to the other sub-space. The reasoning is that when ϵ is 0, following the optimal policy would lead to never crossing the boundary between these two "triangles", meaning that they form two independent sub-spaces. When $\epsilon > 0$, this can happen and the transitions between the two sub-spaces are not small, thus it is an intersting counter-example of using a different kind of decomposition.

As the environments' model dynamics are set by the code, they are known. Thus it is possible to use policy iteration to determine the optimal policy and the optimal state values. This was used to make sure that the tested reinforcement learning algorithms, which model-free, meaning that they are not given the dynamics of the model in input, actually perform optimal actions once convergence is achieved.

4.3 Benchmark DQN agent

The aim of this project is to determine whether state-decomposition techniques can be used to increase the efficiency of reinforcement learning algorithms. Doing so requires comparing the performance of an algorithm where state-decomposition was applied and that of the same algorithm without state-decomposition, while choosing all other parameters to be the same or to be the values that make for the fairest comparison. The number of possible choices to be made in these algorithms is such that providing a fair comparison can be a non-trivial task, as discussed in section-4.4.

The chosen benchmark agent is a basic DQN agent. The value updates occur with discount factor $\gamma = 0.95$. It would be possible to use $\gamma = 1$, as the algorithm in tested in environments with finite length episodes, however, using a slighlty smaller discount factor enhances the stability of the algorithm, giving more consistent results. The agent acts according to an ϵ-greedy policy. This means that at each agent-environment interaction, the agent picks the value that currently has the highest

estimated action value with probability $1 - \epsilon$ or a random action with probability ϵ. ϵ is initialised at 1, meaning that in the first iteration the action is completely random and decays by a factor 0.999 after each update of the action values. ϵ is set to stop decreasing at $\epsilon = 0.01$, this is to ensure that the algorithm doesn't stop exploring by having too small an ϵ. Note that this ϵ is unrelated to the ϵ that represents the probability of transitions between different sub-spaces in 'GridEps'.

Figure-4.4 shows the neural network that approximates the Q-values, which is fully connected and has 4 layers. The input state is fed as an integer to an encoding layer[5], which then feeds to the first fully-connected layer, consisting of 512 neurons with "ReLU"[6] activation function. This is then followed by 2 similar layers, having 256 neurons each. The last layer consists of a number of neurons corresponding to the number of actions in the environment where the agent is being used. Each output of the network corresponds to the action value of an action, thus the activation function was chosen to be linear, that is, the outputs of the layer are equal to the outputs of the neurons. This is to allow the outputs to take any possible values, without upper or lower bounds so that the network is theoretically capable of mapping any state to any possible action-value. The network is updated using "Minimum Square Error" as its loss function, that is, at each update step, the network weights are updated to reduce the function $L = \sum_{i=1}^{n}(y_i - f_\theta(x_i))^2$ where x_i are the inputs, y_i the corresponding outputs and f_θ is the function implemented by the neural network with weights θ. The network updates are done using the optimiser Adam [14], a commonly used optimiser that achieves better performance than gradient descent in many applications, thanks to the use of weight momentums.

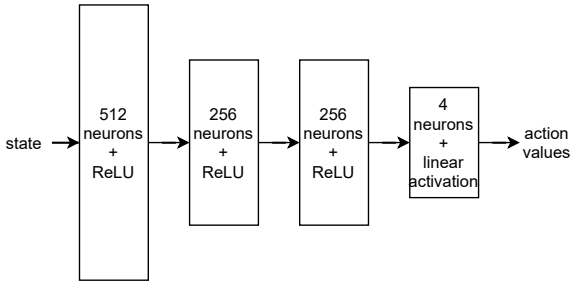


Figure 4.4: The neural network used for Q-values approximation in the baseline DQN algorithm.

Each iteration with the environment is saved in a replay memory as a $\langle S, A, R, S' \rangle$. The maximum size of this is set to be big enough so that all samples are saved, without having to discard them in the later stages of training. After each interaction with the environment, one update step of the neural network is run using a batch of 128 samples, randomly picked from the replay memory.

4.4 Ensuring fair comparison

When comparing the performance obtained using the state-decomposition technique with a benchmark method that doesn't use it, we would like ideally to isolate the effect of the state-decomposition effect, independent to other changes to other parameters. Unfortunately, it is not possible to perfectly isolate this effect, due to the complexity of the relationship between various hyperparameters. However, it is possible to make reasonable choices such that the comparison is as fair as possible. In this project, all hyperparameters were kept constant in all trials. Different experiments that involve different network architectures, such as using 2 neural networks instead of 1 always ensure that the total number of weights in each agent is approximately equal. Other factors such as the encoding method of the state are always kept equal, except when the objective of the experiment is to investigate their own effect on the performance.

[5] This is fixed and pre-programmed and doesn't include any learned parameters.

[6] "ReLU" stands for Rectified Linear Unit. It represents the function $f(x) = \max(0, x)$. Neural networks rely on the non-linear activation functions such as "ReLU" in order to be able to learn non-linear functions.

Chapter 5

Results

This section presents the most important results that I obtained during this project, using different variations of the state-decomposition method and comparing them with the baseline DQN algorithm.

5.1 Evaluation method

This project mainly deals with evaluating the sample efficiency of the state-decomposition method, which is how efficiently the experienced samples (time-steps) are used to learn a policy. The sample efficiency affects how quickly the algorithm can learn an optimal policy and it can be illustrated by plotting the average total reward in each episode versus the number of the episode during training, which shows the improvement in rewards during training, as in Figure-5.2. Another way to illustrate the performance of a reinforcement learning agent is to plot the total reward obtained in the corresponding episode versus the number of experienced samples, as in Figure-5.1. This is the more standard method used to compare the sample efficiency of different algorithms. However, plotting against the number of episodes should give equivalent conclusions,as better performance in one type of graph implies better performance in the other. It is also important to consider the fact that the learning process is stochastic and the performance varies between trials hence the graphs were plotted using values averaged over multiple trials.

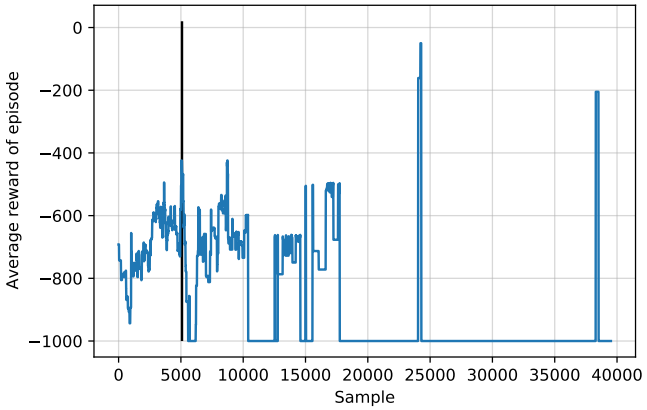


Figure 5.1: An example of an 'average reward' vs 'samples' graph. The black line indicates the length of the shortest trial. Moving to the right of the line, the quality of the graph decreases and the average reward counterintuitively decreases due to the bias of only "bad" trials having such a high sample count.

The main method of presenting the results in this report is using the 'total reward' vs 'episode' graphs. The reasoning for relying less on the 'total reward' vs 'sample' graphs is that the data was

collected in experiments with a fixed number of episodes rather than a fixed number of samples. As the number of samples in each episode varies, each trial has a different number of samples. Thus, an average graph can be plotted only for the length of the shortest trial. Plotting for any longer length would mean that the right section of the graph is an average calculated on fewer trials. This obviously increases its variance, but also introduces a bias. This is because a higher number of samples for the same number of episodes occurs when the episodes are longer, which is when the agent performs worse because successfully completing a task early ends the episode. As a result, if the graph is plotted for more than the length in samples of the shortest trial, the rewards values for high sample counts are negatively biased and as some of the trials last a very short number of samples, the number of reliable samples is small in these graphs.

5.2 Effect of different state representations on the performance.

Although it doesn't necessarily impact the state-decomposition method, it was initially necessary to decide how to encode the states to be fed into the neural network. As a first approach, the state was fed directly as an integer representing the index of the state. This is the most compact encoding of the state, which is what would be used in standard Q-learning, in which the state is used merely as an index in a look-up table rather than as the input of a neural network. Though extremely compact, this representation of the state is very hard to decode for the neural network and its structure does not directly reveal aspects of the state directly[1] and led to difficulties in achieving convergence in a reasonable number of iterations. The lack of physical meaning of the state representation also would make it very hard for the neural network to generalise to unseen states, which is an important feature in large state-spaces. Thus, it was decided the encode the state in three alternative representations:

1. **Compact** The idea is to give a physical meaning to the state representation. The state is given by a vector of values, each representing a different aspect of the physical meaning of the state. In the 'Grid' environment, these are the row in which the Taxi is currently located, the column and its current destination.

2. **One-hot encoding** This aims to make it easier for the neural network to learn to decode the state. The state is in this method simply converted to a one-hot encoded version, which is a vector of length the number of possible states, where all but one elements have value 0, whereas the other one has value 1.

3. **Hybrid encoding** This combines the previous two methods. The destination is represented by a single bit, taking value 0 or 1, while the position of the state is represented by a one-hot-encoded vector. This results in a state representation that is a vector of size one bit greater than the number of possible positions in the environment.

[1] As the state is represented by an integer, this has no physical meaning before decoding. However, the state could be represented for example by a vector of values, each having a physical meaning such as horizontal position, etc.

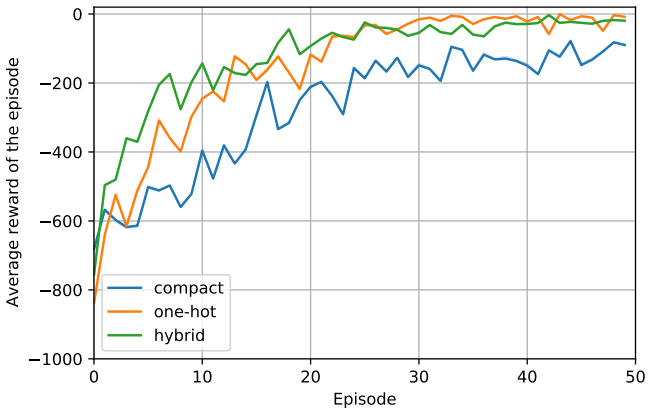


Figure 5.2: Effect of different state encoding methods on the performance of the baseline DQN agent in the 'Grid' environment.

Figure-5.2 shows that in the 'Grid' environment the hybrid encoding achieves the best performance, as the rewards are higher during the training phase, while they are similar upon convergence when compared to standard on-hot encoding. The compact encoding shows the worst performance at all points in time. Thus, it was decided to use the hybrid encoding for all experiments to expedite training times and hence experimentation times. This was actually a major issue during the project, as the training time of the algorithms is long and any increase in performance is welcome as it allows to run more trials to obtain higher quality results and to also run more experiments.

5.3 Effect of varying the neural network's size

The size of the neural network is another factor that influences the speed of learning of deep reinforcement learning algorithms. Figure-5.3 shows that for a DQN agent in the 'Grid' environment, the performance improves as the network size increases, at least for tested range of dimensions. When the network is too small, performance degrades as the network is not complex enough to represent the function that maps from states to action-values of the environment, thus rewards are lower after any number of episodes compared to other networks. In the figure, the three bigger networks are all complex enough to reach convergence, however the biggest one does so quicker. This might be due to increased flexibility and the existence of multiple sets of weights that achieve the same optimal function representation. While this leads to faster convergence and is not a problem in the environments used in this report, it might lead to problems in complex environments that rely on generalisation of unseen states. This is because using an overly big network can easily cause overfitting, thus hurting the generalisation propertiesof the network to unseen states. In this project, it was decided to use networks that are complex enough to capture the dynamics of the environment, while limiting the size so that the project could be relevant in other scenarios where generalisation is imporant. The figure also shows that the difference in performance of 'medium2', which has twice the number of trainable weights of 'medium' is not very significant. I chose to show this comparison because in the environments used in this project, the decomposition divides the state-space into two sub-spaces, which means that there are two NNs corresponding to the sub-spaces instead of the one original NN. To make the comparison fair, I made the total number of trainable parameters approximately equal in any architectures that being directly compared. It is arguable whether this is a fair comparison, as there are more factors affecting the rate of learning, such as the structure of the network, but the fact that the performance hardly varies when doubling the number of trainable parameters demonstrates that this is unlikely to have an important effect on the results.

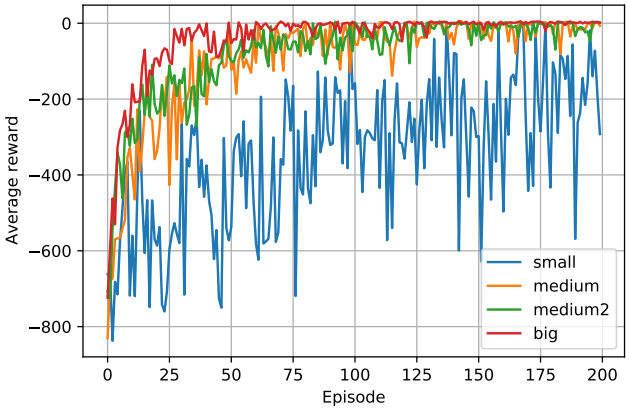


Figure 5.3: Performance of DQN with NNs of different sizes in the Taxi2 environment. 'medium2' has twice the number of trainable weights of 'medium'.

5.4 Initial results with 'Taxi2' and original state decomposition architecture

The original state decomposition architecture proposed in the previous chapter was initially tested in the 'Taxi2' environment. The results are shown in Figure-5.4, where the networks are combined at episode 100. In the first part of training, convergence is not achieved, as the transitions between subspaces are not considered. In the second part of training (starting at episode 100, when the networks are combined), the average reward starts at the same levels as an untrained agent and they as long as a DQN agent to converge. This means that the knowledge learned in the first part of training is 'forgotten' in the second part, thus not contributing to any faster learning in the second stage.

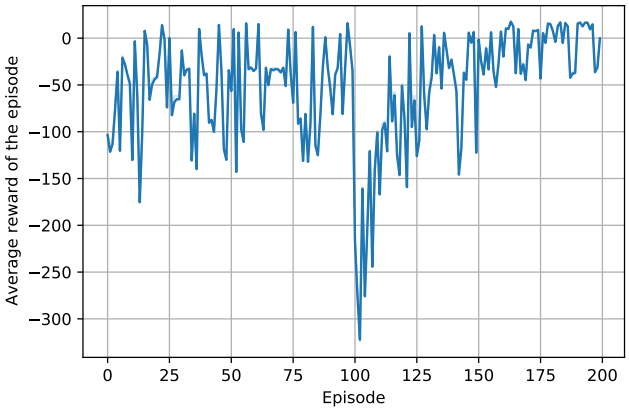


Figure 5.4: Performance of the original two-stage state-decomposition method. The networks are combined at episode 100, where the rewards fall to 'untrained agent' levels and take long to convergence, not any faster than a DQN agent.

5.5 A simple approach to decomposition: 'DeltaSwitch'

Following the failure to obtain positive results using the original method that involves the use of a combining neural network on top of the neural networks corresponding to each one of the

sub-spaces, I decided to experiment with a simpler approach in order to first validate the idea of state-decomposition in the simplest possible application. For this purpose, I introduced a network architecture that I called 'DeltaSwitch'. This consists of one neural network corresponding to each sub-space, as in the originally proposed architecture. The main difference is that the action-values are combined using a simple weighed sum, so that the output action-values of the combined architecture are given by

$$q(s,a) = \sum_{i=1}^{N} \delta q_i(s,a) + (1 - N\delta) \cdot q_{\{j:s\in\mathcal{S}_j\}}(s,a) \tag{5.1}$$

where $q_i(s,a)$ is the action-value of action a given by the network of the sub-space S_i with index i, δ is a small constant that satisfies $\delta \in [0, \frac{1}{N}]$ and N is the number of sub-spaces. This approach is also simpler in the fact that it only consists of one stage of training. This was done to avoid the 'forgetting' that was observed in the original method, which made the first stage of training effectively useless. In fact, all the networks are combined from the start in a predefined manner, as δ is a non-learned constant. The combining by weighed average effectively substitutes the combining neural network and acts as a switch when $\delta = 0$, as it effectively selects the output of the network corresponding to the sub-space of the input space, thus the name 'DeltaSwitch'. The idea of this approach is that it might not be necessary to combine the action-values of the networks in a complicated manner and that doing so using another neurla network might be complicated and lead to stability issues and lack of convergence. The updates of the network are carried out by treating the whole network as a single unit. δ is chosen to be small, so that networks others than the one corresponding to the input state only have a small impact on the output values. As a results, each sample used in an update of the network causes significant change in the weights of the network the correspodning sub-space, while only slightly changing those of other networks. This can be confirmed by deriving the formulas of the updated of weights given the loss function and optimiser, however it is sufficient to understand that those weights that have a smaller effect on the output are updated by a smaller amount in the backpropagation update. As the network is always treated as a whole, the target values required in the update of the network are generated using all the possible networks. For example, if a transition is from a state in sub-space 0 to a state in sub-space 1, even with $\delta = 0$, although the network that is updated most significantly by this sample is the one corresponding to sub-space 0, the target value is generated also using the outputs of the network of sub-space 1. This means that each network is trained using target values generated by other networks, thus the networks are not independent during the training stage. This is required because 'DeltaSwitch' is a one stage method, thus the transitions between must be accounted for since the start of the training. Contrarily, the originally proposed algorithm could initially ignore transitions between sub-space as they were later accounted for through the use of a combining NN.

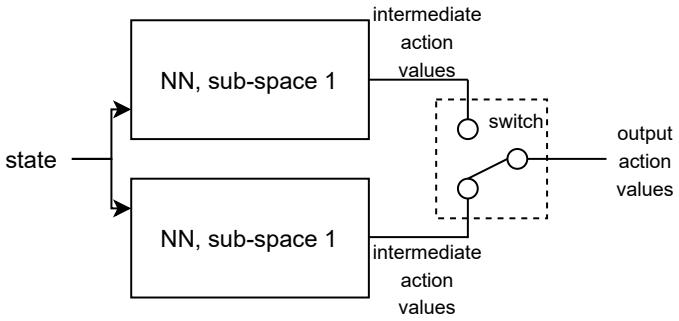


Figure 5.5: 'DeltaSwitch' architecture with $\delta = 0$. In this case, the weighted sum effectively acts as a switch, which selects the output of the network corresponding to the sub-space of the input state.

The first environment where this was tested is the 'Taxi2' environment. In this environment there is no significant difference in performance between DQN and 'DeltaSwitch' with differing δ values, as shown in Figure-5.6.

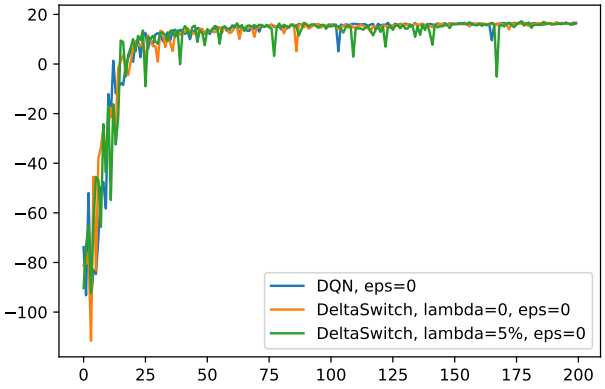


Figure 5.6: Comparison of DQN and DeltaSwitch with $\delta = 0$ in the 'Taxi2' environment.

While this doesn't show that state-decomposition can lead to a performance gain, it shows that it can at least have equivalent performance to a simple DQN agent. The experiment was then repeated in the 'Grid' environment, for easier interpretability of the results. This experiment was repeated in the 'Grid' environment, in which Figure-5.7 shows that 'DeltaSwitch' presents some performance gain compared to DQN when decomposing by destination, as the rewards during the training process are higher. Decomposing by destination means that the possible states are divided in two sub-spaces, corresponding to the two destinations. Decomposing by position means that one sub-space is formed by all those states in which the current position in the upper-left triangle of the Grid, while the other sub-space is formed by those positions in the lower-right triangle.

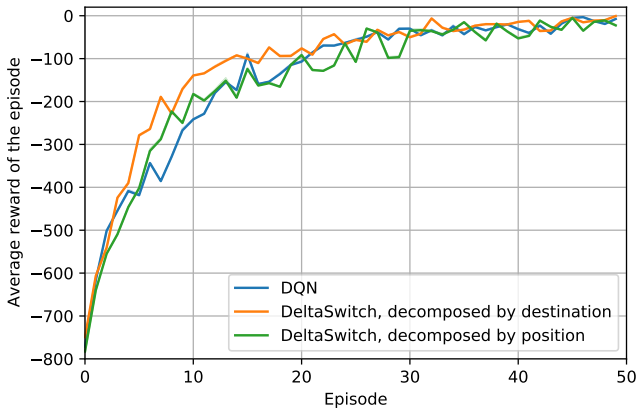


Figure 5.7: Comparison of DQN and DeltaSwitch with $\delta = 0$ in the Grid environment.

A possible reason for the performance gain is that the neural networks of 'DeltaSwitch' represent simpler functions when the decomposition is done by destination. The main difficulty in the function representation is due to states which have the same position having very different action values depending on the destination. The difference given by the value of the destination differs depending on the position. Decomposing by destination is the only of the three methods that makes it so that each network simply deals with assigning action-values to positions, without having to deal with the position. The fact that the neural networks have to learn a significantly simpler target function could justify the increased performance. The experiment was repeated using standard one-hot-encoding, as shown by Figure-A.1 in Appendix-A. Here the differences between DQN and 'DeltaSwitch' decomposed by destination are even more marked, as when decomposing by destination, the networks can

ignore half of the elements of the input vector, while DQN and 'DeltaSwitch' with decomposition by position need to learn to decode the one-hot encoded vector to determine the destination, which is even harder than having the destination given. Additionally, 'DeltaSwitch' by position only has half of the training samples for each network to learn to do so compare to DQN, leading to a further decrease in performance.

This result shows that 'DeltaSwitch' can present some performance gains compared to a normal DQN algorithm in a perfectly decomposable environment. This environment is perfectly decomposable by destination, as the destination never changes, meaning that there are no possible transitions between states having different destinations. It is also perfectly decomposable by position, as when adopting the optimal policy, the agent never moves from one of the triangles to the other, since the destination is always the one closest to the current position. This experiment validates the idea of the state decomposition as having a performance gain in such specific environment mean that it might be possible to also have a performance gain in other environments that are not perfectly decomposable.

For this kind of perfectly decomposable environment there is no reason to set $\delta \neq 0$ as the two sub-spaces are completely independent. Figure-5.8 shows how the performance of 'DeltaSwitch' degrades in the 'Grid' environment when increasing δ. In fact, there is never a reason to set $\delta \neq 0$, as even when the sub-spaces are not completely independent, as communication between them is allowed during training, the action values of the networks are accurate, so the weighted sum should act as a mere switch rather than combining multiple values. The idea of a weighted average was actually initially devised to consider the case when communication between networks does not occur during training[2]. However, we found out that this should be allowed, meaning that there is no reason to set $\delta > 0$.

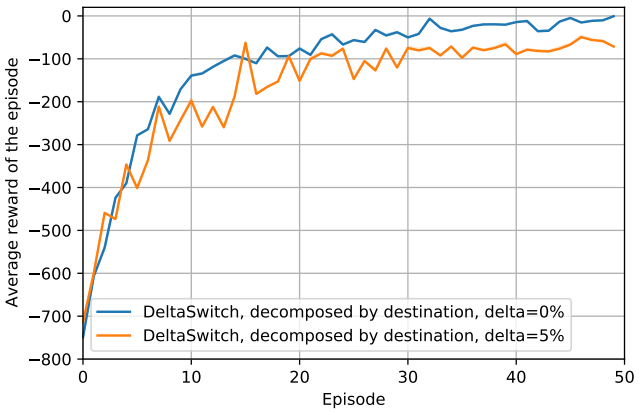
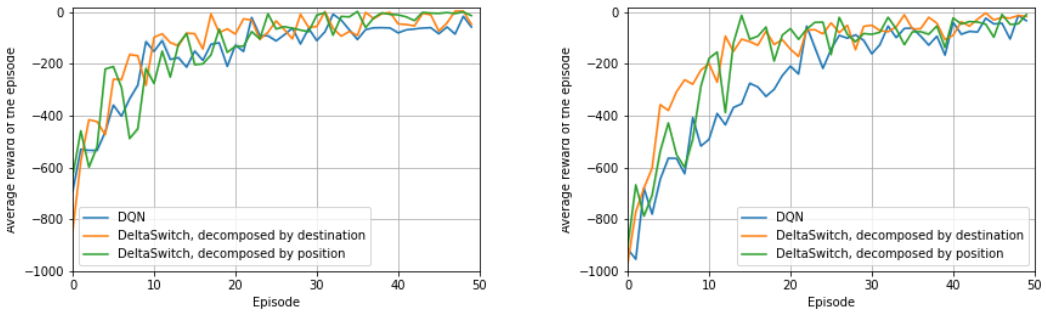


Figure 5.8: Comparison of DeltaSwitch with different δ.

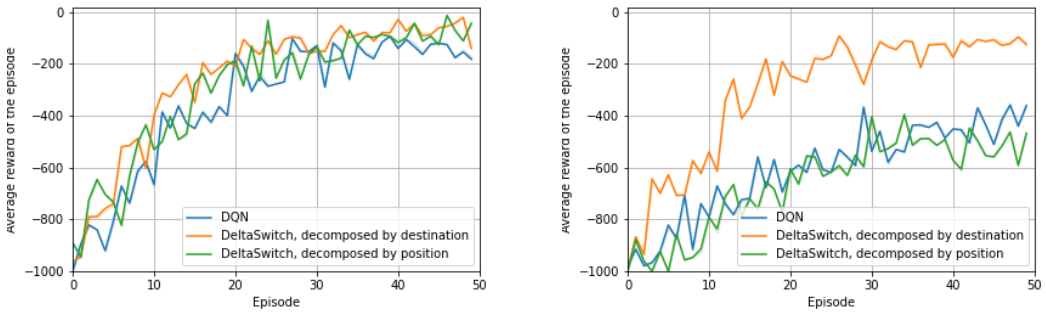
5.5.1 Applying 'DeltaSwitch' to non-prefectly decomposable environment

After determining that a 'DeltaSwitch' agent with $\delta = 0$ can lead to faster training, we shall explore the performance of such an agent in environments that are not perfectly decomposable. This is done by running experiments in the 'GridEps' environment with $\epsilon > 0$. Figure-5.9 compares the performance of training a DQN agent and 'DeltaSwitch' agents, decomposing by destination and by position. It can be seen that for for $\epsilon \leq 3\%$ 'DeltaSwitch' with decomposition by destination learns noticeably faster than the other two algorithms. The difference is especially large for $\epsilon = 3\%$. This might be due to this environment being the harder to solve, giving a greater margin of improvement over DQN. For higher ϵ values, there isn't a statistically relevant difference between

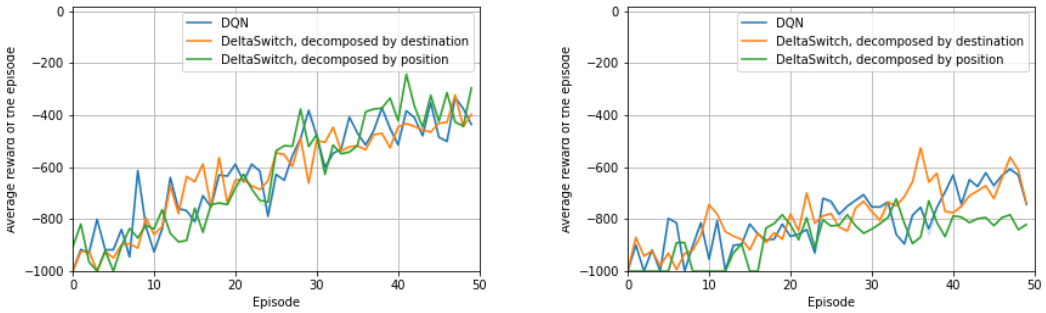
[2]This refers to the generation of target values being allowed to use the networks of any sub-space.



(a) $\delta = 0\%$ (b) $\delta = 1\%$



(c) $\delta = 2\%$ (d) $\delta = 3\%$



(e) $\delta = 4\%$ (f) $\delta = 5\%$

Figure 5.9: DeltaSwitch vs DQN in 'GridEps' with $0 \leq \epsilon \leq 5\%$.

the three algorithms. This is probably due to the fact that the environment is not as decomposable anymore, as the transition proabilities between the sub-spaces are larger.

This is an important result, as it shows that while the state-decomposition method is applicable to perfectly decomposable environments, or environments that are almost fully-decomposable, the performance gain disappears once the transition probabilities between the sub-spaces are too large.

Though these experiments shows that the state-decomposition can be a viable way to reduce training times in certain environments, it does not explain the meachanism in which it does so. The following subsections highlight some of the experiments that were used to shed light on this mechanism.

5.5.2 Effect of using reduced encoding

As I used the 'hybrid' encoding described in a previous subsection, the input to the first layer of the network is a vectors of '0's and '1's of size the number of possible positions in the grid plus one. However, since I am using $\delta = 0$ and each network only contributes to the output when the input state is in the network's sub-space, it is possible to reduce the size of the input vector to only account for the states that are relevant to each network. For example, when decomposing 'Grid' by position, the two networks only deal with 312 and 313 positions instead of the original 625. This means that instead of the original vector of $625 + 1 = 626$ elements, it is sufficient to feed the networks with vectors of $312+1 = 313$ and $313+1 = 314$ elements respectively. I then ran experiments to determine whether such a reduction in input size would improve the performance. Figure-5.10 shows that the performance gain when reducing the input size seems to be marginal and might be caused by the randomness of the trials.

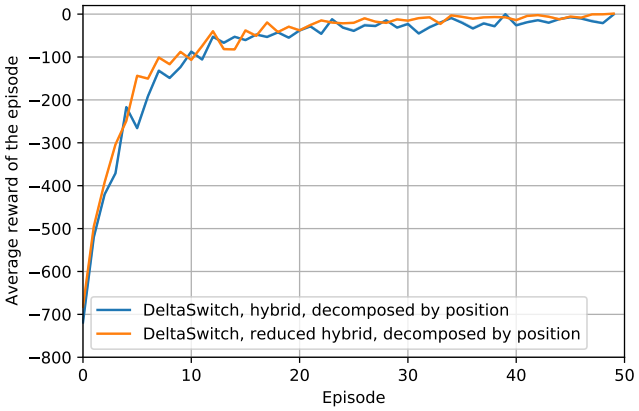


Figure 5.10: Effect of reducing the input size.

This result suggests that the change in size of the input of the network does not significantly affect its abilty to represent a mapping, as long as the number of inputs for which it has to learn this stays constant. Effectively, when using the full input vector size, although the number of inputs is greater, almost half of these are always 0 when the network is active, thus these inputs mathematically have no effect on the network and it is intuitive that the results should be similar when using the reduced input vector.

5.5.3 Effect of adding a wall to separate sub-spaces

The experiments with DeltaSwitch were repeated in a modified version of the 'Grid' environment. This version has a wall that separates the upper-left triangle of the grid from the other triangle. This was done to test the effect of having decomposition upon convergence, versus having decomposition with any possible policy. When there is no wall, the decomposition by position only occurs upon convergence, that is when the optimal policy is being used, as there is no reason why the agent would act to cross the boundary between the two sub-spaces. However, it is physically possible to cross this boundary and this is very likely to occur during the training stage. Conversely, when a wall is present, it is not physically possible to cross this boundary, regardless of the policy.

It is intuitive that when the wall is present, the learning process is faster, as it impedes the agent from moving further from the its destination. It is interesting to notice that as shown in Figure-5.11, the performance of 'DeltaSwitch' and the benchmark DQN algorithm is equivalent in this environment. Any differences in the figure are small enough to be attribute to the randomness of the training, as they are not consistent throughout the graph, with the lines crossing each other multiple times.

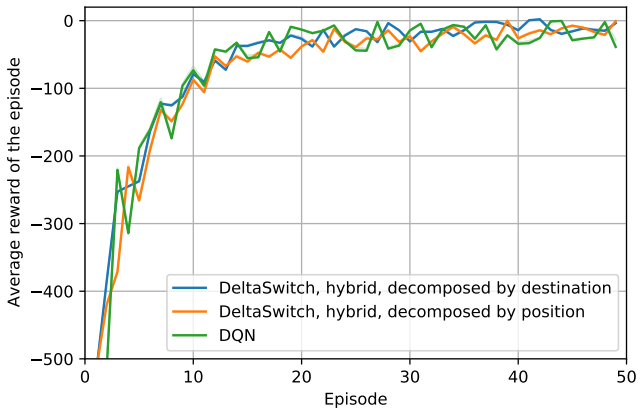


Figure 5.11: Performance of DQN and DeltaSwitch with different decomposition in the 'Grid' environment with wall.

Whereas in the environment without a wall, 'DeltaSwitch' with decomposition by destination presents a performance gain, this does not occur when there is a wall. This agrees with the theory that the performance gain of 'DeltaSwitch' with decomposition by destination is due to the networks having to learn simpler target functions. When the wall is present, states in which the destination is far, that is, states that are on the wrong side of the wall, are never visited. As a result, for each position on the grid there is only one possible destination, meaning that the function to be learned by the network is already simplied, even for DQN and that applying the decomposition does not simplfy it further, and thus the performance is equivalent for the three methods.
(COULD TALK ABOUT GridEps2 but it's not necessary)

5.6 A different approach based on transfer learning

The idea of decomposition exploits the fact that the optimal policy to be followed in a perfectly decomposable environment should be similar to that of a similar environment, having a small probability of transitions between states of the different sub-spaces. Another class of techniques that leverages similarity between environments is transfer learning. Using transfer learning in reinforcement learning means to use the knowledge obtained by having trained an RL agent in a certain environment in order to more efficiently learn in a new similar environment. The 'DeltaSwitch' algorithm demonstrates how it is possible to achieve some performance gain by state-decomposition and it does so in one stage of training. The original idea was to first train multiple agents that deal with each of the sub-spaces and then combine them using another neural network. This idea is very similar to training an agent in the perfectly decomposable environment and then using its knowledge, more efficienlty learn in the more complex environment. Effectively this two stage approach is transfer learning from a simpler fully decomposable environment. Thus, exploring transfer learning techniques could give insight about possible ways to apply state-decomposition using a 2-stage architecture.

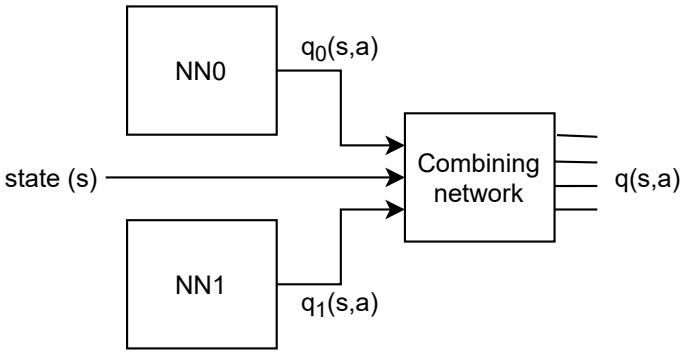


Figure 5.12: Combining 'NN0' and 'NN1' using another NN.

The following methods were all tested by using the two networks from a 'DeltaSwitch' agent that was previously trained in a 'Grid' environment, decomposed by position [3], which gives two neural networks, called 'NN0' and 'NN1', which correspond to the two sub-spaces. The transfer learning techniques were then tested in a 'GridEps2' environment.

The first approach was to combine 'NN0' and 'NN1' using another neural network. The weights of 'NN0' and 'NN1' are set to be fixed. The combining network is also fed the input state. The idea is that this network can act as a clever switch instead of the 'rudimentary' switch used in 'DeltaSwitch', in order to take into account the transition between sub-spaces. This algorithm gives the results in Figure-5.13, which shows that this algorithm, despite using 'NN0' and 'NN1' which were previously trained, achieves lower rewards than a standard DQN algorithm for both values of ϵ. This gap increases for higher values of ϵ. This means that this method is unsuccessful in leveraging the knowledge obtained from 'NN0' and 'NN1'

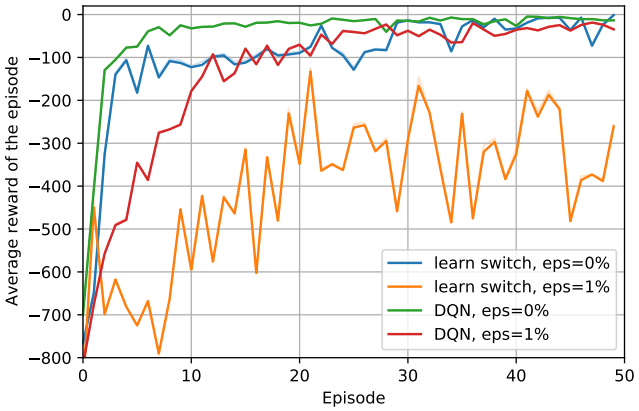


Figure 5.13: Performance of combining 'NN0' and 'NN1' using another NN that is also fed the current state as input in a 'GridEps' environment with $\epsilon = 0\%,\ 1\%$.

As the ϵ of the 'GridEps2' environment is set to be small, it is fair to assume that directly using the original 'DeltaSwitch' agent that 'NN0' and 'NN1' are taken from should give near-optimal performance. Thus, at least in the start, the transfer learning method should achieve much higher rewards than a standard DQN. This was achieved by 'forcing' the initialisation of the combining network, whereas before this was randomly initialised. This is done by training the combining NN in a supervised manner so that the combined network ('NN0','NN1' and combining network) is initialised to

[3] This kind of decomposition was chosen as it has better performance in the 'GridEps2' environment (COULD SHOW IN APPENDIX).

behave exactly as the 'DeltaSwitch' of 'NN0' and 'NN1'. The training samples have as inputs all the possible states and as outputs the correspodning outputs of 'DeltaSwitch'. After training for multiple epochs[4], with MSE as the loss function, the combined network can be utilised in the environment as before and RL process can be started. The initial exploration rate is set to be low[5] as it is assumed that it is not needed to perform lots of exploration. Figure-5.14 shows the results of doing so with different NN sizes. The performance is much better than without initialisation in the start. Using the smallest network does not allow to capture the environment's dynamics and fails to converge. Only the biggest NN allows to achieve performance comparable to DQN in the last part of training and in this case the network used is actually bigger than that in DQN.

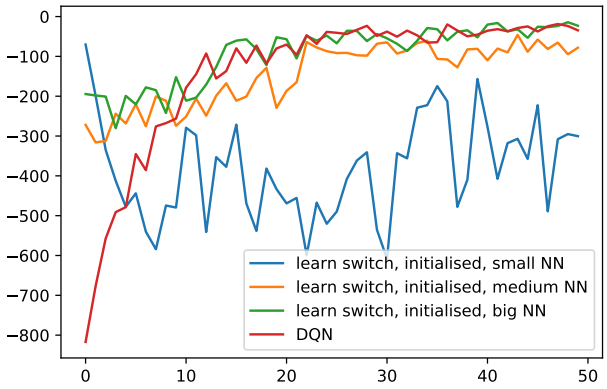


Figure 5.14: Performance of combining 'NN0' and 'NN1' with another NN when initialising the combined network to behave as the original 'DeltaSwitch'. Tested with different network sizes

Requiring a combining network of a similar size to that of the DQN agent means that the function that this NN has to represent is similarly difficult to that of the DQN's network. Thus it seems that the initialisation of the network only serves the purpose of limiting negative rewards at the start of training, but it doesn't have a lasting effect in making the learning easier. The issue could be that, though Q-learning is guaranteed to converge after sufficient amounts of exploration, it always finds the optimal policy, so it is likely to completely discard the initial policy provided when initialising the combined network, causing so called "forgetting". I thus experimented with SARSA, as its updates are given by the formula

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{5.2}$$

that is, SARSA is an on-policy algorithm, meaning that the action values of the states are updated using the action values that were actually taken (UNCLEAR). My intuition is that doing so is more likely to preserve the initial policy provided by the initialisation of the combined network, which would increase the speed of learning at the expense of the performance upon convergence. However, using SARSA failed to convergence, thus I tried with expected SARSA, which has the updates

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\{R_{t+1} + \gamma \, \mathrm{E}_\pi[Q(S_{t+1}, A_{t+1})] - Q(S_t, A_t)\} \tag{5.3}$$

where π is the policy at the time of the update. Figure-5.15 shows that using expected SARSA seems to improve the initial performance, as rewards never fall below -200. However, in the long term the performance is still equivalent to DQN, as this method does not reach convergence earlier than DQN, thus the problem of "forgetting" is not solved.

[4] In the experiment, the network was trained for 1000 epochs, in order to ensure an accurate initialisation. However, such high accuracy might not be needed.

[5] The ϵ of ϵ-greedy

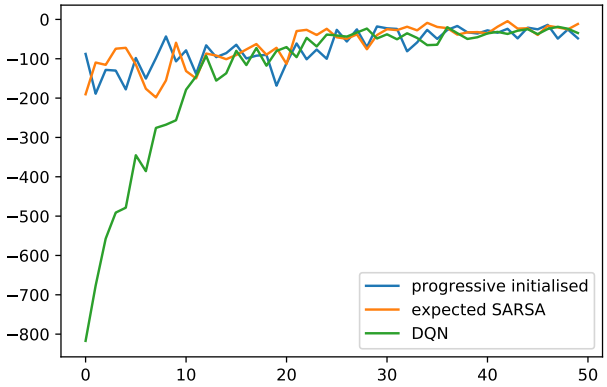


Figure 5.15: Comparison of using Expected SARSA after combining, progressive combined network and standard DQN.

Subsequently, I experimented with the idea of 'progressive neural networks' [23]. This is a kind of neural network that was devised for transfer learning, as it shows better performance in transferring knowledge between tasks and it avoids "forgetting". The progressive neural network was devised for learning successive related tasks. In the first task, a neural network is used as normal. Then, in the second task, the neural network is formed by the neural network trained in the first task with its weights frozen and a new 'column' of layers. Each layer in the new column receives in input the outputs of the previous activation layer of the same column and the outputs of the activation layer of the network of the previous task. This combined network is then trained in the new task and the process is repeated for all the tasks, each new column receiving in input the outputs of all the previously trained networks. Figure-5.16 shows the structure of a progressive network used on a third task when transferring knowledge from two previously trained tasks.

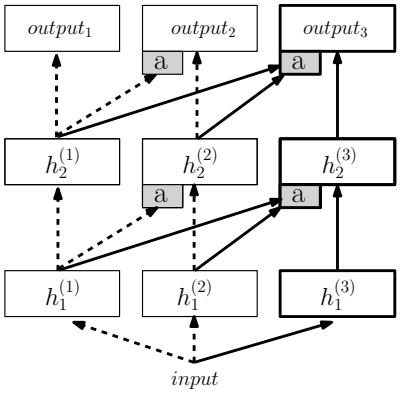


Figure 5.16: Structure of progressive network when training the third task. Figure taken from [23].

The architecture that I propose is very similar. Instead of connecting the networks one after the other, I simply connect one new network to all the previously trained networks that correspond to the sub-spaces. That is, I take 'NN0' and 'NN1', I freeze their weights and I connect the outputs of all their layers to every input of a new network. This new network has thus the same number of layers as 'NN0' and 'NN1'. The architecture is shown in Figure-5.17. This was initially tested with random initialisation of the combined network, leading to slow convergence and worse performance than DQN. However, when initalising the combined network similarly to what was previously done in the other combining methodology, the performance is similar to what was achieved in the other

methodology using expected SARSA, as shown in Figure-5.15. Thus, this technique also seems to not provide an important transfer of knowledge and solve "forgetting".

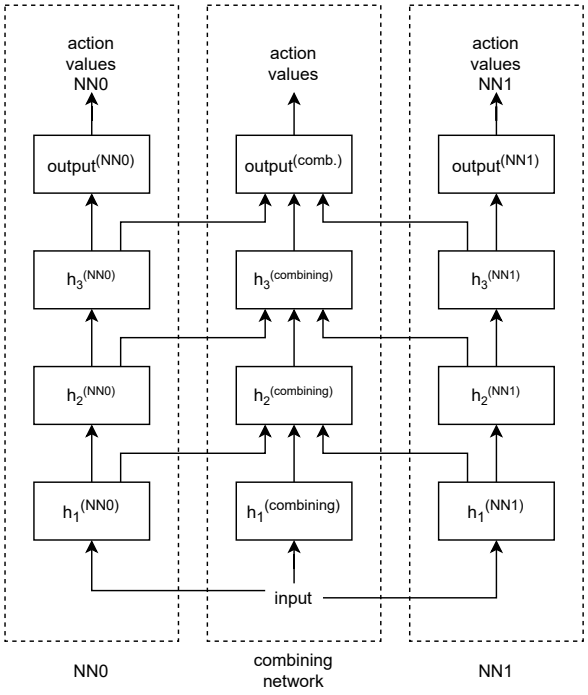


Figure 5.17: Modified progressive neural network. The layers of the combining neural network receive as inputs the respective outputs of the previous layers of frozen 'NN0', frozen 'NN1' and the combining network itself. h indicates hidden layers of the network.

These transfer learning techniques could be used to build a two-stage state-decomposition architecture in which in the first stage we consider the transitions within sub-spaces and in the second stage we apply transfer learning to learn the full environment, including all transitions. This was attempted without success, as providing a good initialisation for the second stage requires an almost optimal policy in the first stage, which takes many episodes to reach, thus not providing any benefit over DQN.

5.7 Techniques to learn the 'switch'

While the transfer learning techniques in the previous subsection show superior performance to DQN, this is an unfair comparison, as they rely on the previously trained networks 'NN0' and 'NN1'. Moreover, the better performance seems to be due to a good initialisation which doesn't translate to a successful transfer of knowledge and earlier convergence. However, the two combining techniques can be modified to work in one stage, thus simply providing a decomposition rather than doing transfer learning. This, provides a more flexible architecture than 'DeltaSwitch' that could be used even when the networks of the sub-spaces don't account for all possible states, as could occur when using different types of state-decompositions as described in the next chapter.

The first proposed technique is to use a combination neural network right from the start, instead of training 'NN0' and 'NN1' first using a 'DeltaSwitch' agent. 'NN0' and 'NN1' are forced to learn the action values corresponding to the decomposed sub-spaces by including their outputs in the the combined network's output and thus in the loss function. Understanding this requires some detail on how target values are normally generated. In the standard DQN algorithm, at each update step, a batch of samples is selected from the replay memory. Each sample provides information about

what the target value [6] of a state-action pair. The target value of those state-action pairs that are not in any of the batch's samples are simply set to be equal to current output of the network for the corresponding states. This way, the network tends to improve for the state-action pairs in the batch's samples, while penalising changes in other pairs. A similar method is used to deal with the fact that only certain samples should update 'NN0' and the others should update 'NN1'. For those transitions samples that start in a state that is in sub-space 0, the target value for the output of 'NN0' is set as normal using the formula of Q-learning updates, while the target value of 'NN1' is simply set to be what 'NN1' currently outputs for the considered state. The same is true the other way around. The output of the combining network is treated the same way as the output of the network in DQN. This methodology allows the combining network to learn the true action-values for all states, while 'NN0' and 'NN1' learn them for the states in the assigned sub-space. However, as seen in the previous subsection, the combining neural network did not seem to be able to make use of the information provided by 'NN0' and 'NN1'. As here 'NN0' and 'NN1' are not provided pre-trained, one can only expect the results to be worse. This is confirmed by Figure-5.18[7], which shows that in 'GridEps2' with $\epsilon = 1\%$, this architecture performs worse than standard DQN.

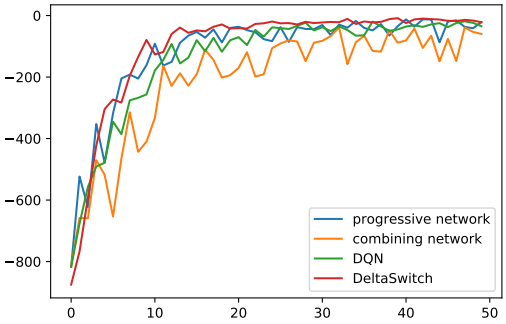


Figure 5.18: Performance of progressive network and combining network when 'NN0' and 'NN1' are not pre-trained. Standard DQN and DeltaSwitch are shown for comparison.

The second proposed technique is based on the modified progressive network from the previous subsection. 'NN0' and 'NN1' instead of being pre-trained and having frozen weights are trained in a similar fashion as what was done in the first proposed tehcnique in this subsection. That is, the progressive network is composed of three columns, 'NN0', 'NN1' and the combining network which at every layer also receives the outputs of the previous layer in 'NN0' and 'NN1'. This is essentially, the same as the architecture shown in shown by Figure-5.17, but 'NN0' and 'NN1' are now not frozen. The only difference between this method and the first proposed method is given by the connections between 'NN0', 'NN1' and the combining network, but the way the target values are generated is completely equivalent. This second technique achieves better results than the previous one, probably due to the greater number of connections between the networks, acting at all the layers, which allows for a better flow of information from 'NN0' and 'NN1'. Figure-5.18 shows that the performance is slightly better than DQN and slightly worse the 'DeltaSwitch'. This means that this technique is capable of leveraging the state decomposition in order to speed up the training. In the previous technique, the only connection between 'NN0', 'NN1' and the combining network was given by the outputs of 'NN0' and 'NN1'. This makes it easy for the combining network to completely ignore 'NN0' and 'NN1', thus making the training of 'NN0' and 'NN1' something tha complicates the training without helping it in any way, thus slowing it down.

At this point one could ask why even use these more complicated architectures when 'DeltaSwitch' seems to provide better performance. The reason is that while 'DeltaSwitch' works well when using

[6] Target value refers to the values used at the output to calculated the loss function, as the neural network is trained in a supervised manner.

[7] All the experiments in this figure were run by applying decomposition by position.

the kind of state-decompositions that we consider in this project, there are also different way to decompose a state-space that would require the use of a combination network. For example, consider having a state-space represented by a $2n$-dimensional vector, composed of 2 n-dimensional vectors. The dynamics of the model are such that transitions in which both sub-vectors change are very rare. Using the original decomposition method, this would not decompose. However, using alternative decomposition methods we could for example decompose by simply splitting the vector into the two sub-vectors, such as each of two neural networks deals with changes in one of the sub-vectors.

Chapter 6

Applicability of the proposed technique

The previous section shows that given certain conditions, the state-decomposition method can give a performance gain in terms of the speed of learning of a reinforcement learning agent. Additionally, such methods are able to leverage information from simpler environments to learn faster in a more complicated version of said environments. This secion aims to provide a more real-life like examples to illustrate to potential of the state-decomposition method in scenarios where using a single global agent could be deemed wasteful, given the better efficeincy of the proposed methods.

In general, techniques based on the state-decomposition method could be developed for any kind of problem that is decomposable. Usually, such a characteristic occurs in systems composed of multiple sub-systems, where the various components seldom interact with each other. It is useful to discuss a possible application of the methodology to better show how such a technique could be applied to a real-life scenario.

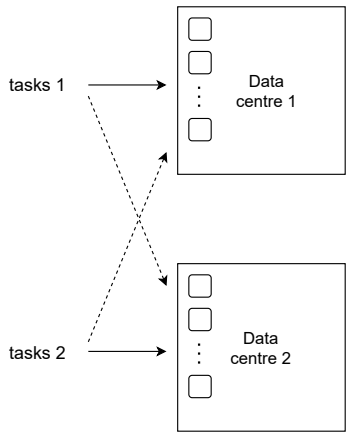


Figure 6.1: Allocation of resources problem for two interconnected data centres.

Consider a scenario where we have two data centres, each of which has multiple computers, as showing in Figure-6.1. Each of these data centres receives tasks (jobs) that can be assigned to the computers and completed. Each of these jobs requires a random time to process on its allocated machine. At each time-step there can be no task or 1 task arriving to one of the data centres. The resource allocation problem deals with assigning these tasks to machines in the two data centres in order to minimise the average waiting time to completion of the tasks. Most of the time it is most efficient for the data-centre that receives the task to fullfil it, as sending the task to the other data-centres incurs in additional delays. However, sometimes a data-centre is busy enough to justify the

additional delay incurred in trasferring tasks between data-centres. This problem can be carefully formulated as a Markov Decision Process and solved using reinforcement learning. The MDP can be defined as follows:

- The state of the MDP indicates the number of tasks waiting for each machine, that is, a computer in a data centre. It also indicates the presence of eventual incoming tasks in the data centres. Thus the state can be represented by a vector having an element for each machine, which take integer values indicating the number of jobs waiting for processing in each of the machines. The vector would also include additional two elements, taking values 0 or 1, indicating the presence of new incoming tasks in each of the data centres.
- The actions to be taken by the controller are to assign the tasks to the machines. At each time-step, the controller can allocate the received task to any machine, either in the local or in the remote data centre.
- The rewards can be positive values that occur when tasks are completed. In order to encourage resolution of a task in a local machine one the rewards can be set to be higher when the task is completed in the data centre that received the task than when it is completed by the other data centre.

If the difference of the rewards upon completion of tasks between local and remote data centres is set to be very high, the controller will allocate most tasks locally. The fact that the controller is unlikely to assign tasks to the remote data centre makes it so that the equivalent transitions are unlikely to happen and that the state-space can be decomposed accordingly.

The simplest way of applying a state-decomposition in this scenario would be with a 'DeltaSwitch' agent having two neural networks, one for when a task is received by one data centre and another NN for when a task is received by the other data centre. Though this might be effective, it does not use the decomposition previously discussed based on decomposing by 'ignoring' state transitions below a certain threshold. In this case, an intuitive decomposition of treating two relatively separate problems as such is used.

In the case where the difference in rewards between completing a task locally or remotely is infinite, the tasks would always be solved locally. In this case the problem of deciding which data centre the task should be processed in is inexistant. In this case we could simply have two separate agents for the two data centres that learn how to assign tasks within the data centres, without dealing with their interconnection. Howver, as the difference in rewards decreases, it becomes optimal to use the interconnection. The 'learn the switch' and transfer learning techniques discussed in the previous chapter in this case would be useful. The networks trained to assign tasks internally within the two data-centres can be combined in order to learn a global agent that can assign tasks in both in the most optimal manner. This architectures is illustrated in Figure-6.2.

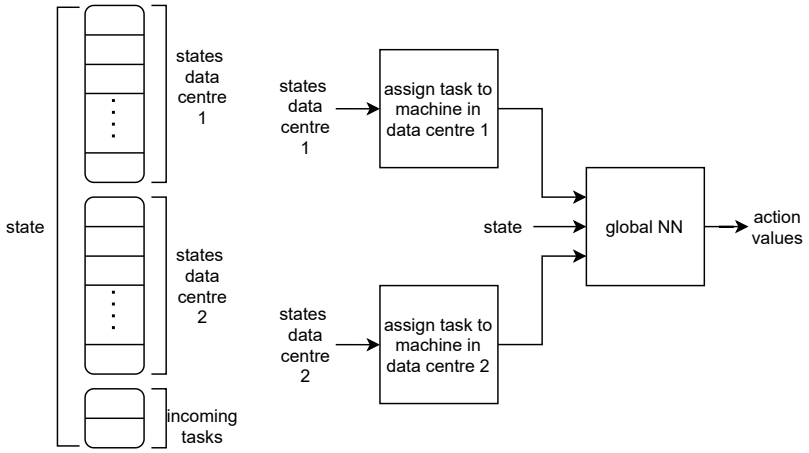


Figure 6.2: Possible state-decomposition agent.

This last methodology is very similar to what would be done in a hierarchical architecture. This would have an agent that explicitely decides whether a task should be assigned to data centre 1 or 2. Given this knowledge a sub-agent trained to decide how to assign the task internally within the data centre would be used and the outputs of these two would be combined to give the full action description. The difference with the state-decomposition method is that while in the latter the full-action description is given by the combination network and all other components merely provide implicit information to it, in the hierarchical approach the action description is formed explicitely by combining the outputs of multiple 'layers' of sub-agents. Additionally, the hierarchical approach deals with the problem on 'high level' to 'low level' approach, in which 'high level' actions are decided first and 'low level' actions are determined using the 'high level' actions, while the state-decomposition approach feeds the knowledge of the possible 'low level' agents to a global agent that deals with all levels of action.

Thought it was shown that the proposed state-decomposition techniques can be beneficial in terms of sample efficiency, there is an increase in complexity of the design of the agent. The main issue is that the state-transition matrix is not normally known in applications that use reinforcement learning, meaning that if the decomposition is not "intuitive" as seen in previous examples, this would have to be estimated before being able to apply state-decomposition. Additionally, there are cases in which there might be different types of possible decompositions and there does not seem to be a systematic way to predict which one is most efficient. Lastly, while it was shown that the state-decomposition techniques achieve performane gains in certain environment, this is not guaranteed for any scenario, thus the extra design effore might not be rewarded by increased efficiency is some cases.

Chapter 7

Conclusions and Further Work

7.1 Findings

The performance of different state-decompoition architectures was tested in various environemnts. It was found that the initally proposed architecture which is trained in two stages fails to transifer knowledge from the first to the second stage. The most positive results were obtained with the 'DeltaSwitch' architecture with $\delta = 0$, which simply assigns a different neural network to each of the decomposed sub-spaces, but allows to train each neural network using target values generated by the other neural networks. This obtains a performance gain over DQN in various environments, as it experiences higher rewards during training, though not necessarily reaching convergence faster. It was identified that the most likely way in which 'DeltaSwitch' improves performance over DQN is by simplifying the functions that the neural networks aim to learn.

Two transfer learning techniques were also tested, in order to gain insight on what could be possible techniques to use in two-stage state-decomposition architectures. One of them uses a neural network to combine the netoworks of the sub-spaces, while the other combines them using the idea of a progressive neural network. We were successful in using the pretrained networks to provide a good initialisation in the training environment, however, the rate of convergence of these techniques was the same as for DQN. It was attempted to apply these techniques to a two-stage decomposition approach, however this proved to be difficult as a good intialisation of the second stage required a long training time in the first stage.

The two transfer learning techniques inspired two other techniques having only one-stage. These are basically the transfer learning techniques in which the pretrained networks are not pretrained, but are trained simultanously with the combining network. The architecture using a simple combining network did not present any performance gain over DQN, while the one using a progressive network architecture did.

It was thus found that the state-decompositon technique can lead to "cheaper" training, meaning that the rewards are higher during the training process. The 'DeltaSwitch' agent showed the best performance in my tests, while the progressive network architecture showed slightly lower performance but it can be used with a wider range of decompositions. A possible use is in the allocation of tasks to interconnected data-centres. A performance gain was only achieved using one stage approaches, however, two-stage approaches could be successful if a way to successfully transfer knowledge to the second stage is determined.

7.2 Discussion

One limitation of this project is that due to timing and computational constraints, the state-decomposition technique was only tested in relatively simple environments with small state-spaces. As this technique aims to aid in scenarios with very large state-spaces, it would be useful to test it in such environments. However, I predict that the performance gain of state-decomposition could increase further for very large state-spaces. The reason is that the environments used in this project were small enough so that all states were likely to be visited multiple times and the neural networks approximating Q-values would not have to massively rely on generalisation. However, the generalisation properties of neural networks are increasingly important as the size of the state-space increases, as it becomes harder to visit all the states during training. As the state-decomposition methods train a sub-agent for each sub-space, these agents only have to generalise over the smaller sub-spaces. It might be that this effect outweights the fact that there are fewer samples for each sub-space than for the whole state-space, leading to better generalisation and thus faster convergence.

Additionally, for the tested scenarios a higher performance was achieved using bigger neural networks, due to the small size of the state-spaces which leads to better learning with massive neurla networks that simply memorize values for each possible state, thus overfitting. However, in scenarios with big state-spaces that rely on generalisation, overfitting is very harmful. The state-decomposition technique by simplifying the functions that the neural networks have to learn, could allow the use of smaller neural networks that are less likely to overfit.

A limitation of the state-decomposition method, at least in the explored versions, is that it fails to exploit similarities between sub-spaces. This is because while diving the state-space into smaller sub-spaces, it also reduces the number of available training samples for each sub-space, as each sample is only used by one sub-space. However, some of these samples could provide information for many similar sub-spaces. For example, in the 'Grid' environment there are pair of states that are specular to each other, as in their action-values are almost identical for swapped actions. A more efficient technique would be able to identify these similarities and further improve sample-efficiency.

Another problem is that while it was shown that the state-decomposition method can lead to performance gains in certain environments, using certain network architectures and and hyperparameters, this doesn't necessarily imply that this would also occur with different design choices. For example, all experiments shown in this report used an ϵ-greedy policy and $\gamma = 0.95$, however, different values of these might affect the results.

A limitation of the experimental technique is that some of the results were obtained using a relatively small number of samples (ranging from 20 to 100), which was not always consistent due to timing and computational power constraints. However, the results are considered statistically relevant and it is unlikely that running further trials would have affected to conclusions of this report.

Finding the state-transition matrix The first step in the state-decomposition method is to obtain the state-transition matrix. Since we are testing the algorithm in software simulations, the state-transition matrix can be easily obtained from the source code of the environments and it can be considered as given. This is usually not the case, so at a later stage we shall test methods to approximate the state-transition matrix. This can be done by interacting with the environment and sampling the $\langle S, A, R, S' \rangle$ tuples that are generated. Since the state-transition probabilities are dependent on the action, the state-transition matrix is dependent on the chosen policy. As we are conducting this before the training and we have no information about the optimal policy, we choose to use the most general possible policy to obtain the state-transition table, that is, the actions are all chosen with equal probability independently of the current state. The number of iterations required to construct the state-transition matrix is dependant on the number of states, actions, the probability of visiting each different state and the required level of accuracy of the state-transition matrix estimate.

7.3 Extensions

Compression and embeddings of states The state-decomposition algorithm divides the original state-space of the system into smaller groups of states. It can be supposed that the variance of the states within these sub-spaces is smaller than in the original state space. Thus, the states that are input in the sub-spaces' agents could be successfully compressed into a more compact representation which could positively affect the performance of the algorithm. More generally, it could be beneficial to learn custom embeddings for the states of each sub-space, having the aim of better generalisation rather than state-space reduction.

Applying state-decomposition to continuous state-spaces The state-decomposition method is only applicable to discrete state spaces. A possible extension would investigate methods to decompose continuous state-spaces. A state-transition matrix could be obtained by discretising the states and the state-space would be decomposed accordingly, while the agents could still be fed the original continuous states as input.

Different types of decomposition This project mostly focused on decomposition using transition probabilities, however this is not the only possible kind of decomposition, as seen in the data centres example. Example of different decompositions are:

- **Applying threshold to transitions between whole sub-spaces** The original state-decomposition method in this project consists of temporarily ignoring improbable state transitions by applying a threshold to the state-transition matrix. There are scenarios in which groups of states are connected by state-transitions that have a high probability but it would still be useful to consider them as separate sub-spaces. Consider for example Figure-7.1, where an MDP is composed of nearly independent sub-spaces that are only connected by the blue transitions. In such a scenario it could be useful to determine the sub-spaces by limiting the number of possible non-zero transition probabilities between any two sub-spaces rather than applying a probability threshold to every single transition. Alternatively, we could apply a threshold to transition probabilities between sub-spaces instead of single states.
- **Decomposing along the length of the state vector** This could be useful in a case like the example with data-centres, in which different transitions are only likely to affect different sections of the input state's representation vector. Thus, different neural networks could deal with each of these sections separately.

Sharing information between sub-agents Consider a system composed of many identical sub-systems. The proposed decomposition method would be able to exploit the structure of the system to learn more efficiently. However, in this case, it would be possible to also exploit the similarity of the sub-systems in order to be even more efficient. This is because in this case, the sub-agents would basically be identical, meaning that the samples belonging to one sub-space could also be used to train the agents of the other sub-spaces. This is an extreme scenario, however, there are many common environemnts in which the sub-systems are similar if not identical. Studying ways to share learned information between sub-agents to exploit the similarity of the sub-spaces could further increasing the sample efficiency of training.

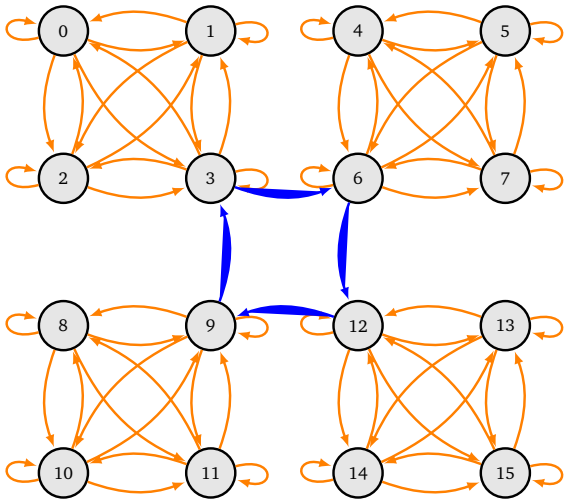


Figure 7.1: MDP composed of smaller sub-spaces of states connected by few but probable transitions.

Using state-decomposition for automatic sub-goal discovery The idea of state-decomposition, especially when implemented by considering transitions between whole sub-spaces could be applied to the automatic discovery of sub-goals in hierarchical reinforcement learning. A similar idea has been adopted by [37] and [15], in which the graph of the MDP is divided into sub-spaces by cutting through the least possible number of state-transitions. The intuition is that sub-goals, such as driving towards the passenger in "Taxi-v3" or driving to the destination after picking it up, are often connected by bottleneck states, such as states 3, 6, 9 and 12 in Figure-7.1.

Proving mathematical bounds It could be useful to derive mathematical bounds for the rate of convergence of different state-decomposition algorithm. This could provide theorical guidance for more efficient experimentation of new techniques.

Applying state-decomposition to policy gradient algorithms The state-decomposition method was mostly applied in this project to methods based on deep Q-learning. However, state-decomposition is not limited to DQN and it could for example be applied to policy gradient methods.

Bibliography

[1] Cloud tensor processing units (TPUs). pages 14

[2] Google AI blog: Scalable deep reinforcement learning for robotic manipulation. pages 2

[3] Markov decision process. Page Version ID: 1002665350. pages 5

[4] OpenAI gym: the CartPole-v0 environment. pages 15

[5] TensorFlow. pages 14

[6] Mehran Asadi and Manfred Huber. State space reduction for hierarchical reinforcement learning. 2004. pages 8

[7] Ravi B Kiran, Ibrahim Sobh, Talpaert Victor, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *arXiv.org*, Feb 2020. pages 2

[8] Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. 13(1):41–77. pages 8

[9] R. Bellman. Dynamic programming. 153(3731):34–37. pages 6

[10] Thomas L. Dean, Robert Givan, and Sonia Leach. Model Reduction Techniques for Computing Approximately Optimal Solutions for Markov Decision Processes. *arXiv e-prints*, page arXiv:1302.1533, February 2013. pages 8

[11] Thomas G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. pages 8

[12] Chris Gaskett, David Wettergreen, and Alexander Zelinsky. Q-learning in continuous state and action spaces. In Norman Foo, editor, *Advanced Topics in Artificial Intelligence*, volume 1747, pages 417–428. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science. pages 8

[13] David Ha and Jürgen Schmidhuber. World Models. *arXiv e-prints*, page arXiv:1803.10122, March 2018. pages 8, 9

[14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. pages 19

[15] Ishai Menache, Shie Mannor, and Nahum Shimkin. Q-cut—dynamic discovery of sub-goals in reinforcement learning. In Tapio Elomaa, Heikki Mannila, and Hannu Toivonen, editors, *Machine Learning: ECML 2002*, Lecture Notes in Computer Science, pages 295–306. Springer. pages 42

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv e-prints*, page arXiv:1312.5602, December 2013. pages 7, 8

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. 518(7540):529–533. pages 7

[18] Thomas M. Moerland, Joost Broekens, and Catholijn M. Jonker. Model-based reinforcement learning: A survey. pages 8

[19] OpenAI. Gym: A toolkit for developing and comparing reinforcement learning algorithms. pages 14

[20] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. page 7. pages 8

[21] Romain Paulus, Caiming Xiong, and Richard Socher. A deep reinforced model for abstractive summarization, November 2017. pages 2

[22] Paul J. Pritz, Liang Ma, and Kin K. Leung. Jointly-Trained State-Action Embedding for Efficient Reinforcement Learning. *arXiv e-prints*, page arXiv:2010.04444, October 2020. pages 8

[23] Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. pages 32

[24] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. pages 2, 9

[25] Richard S. Sutton. Dyna, an integrated architecture for learning, planning, and reacting. 2(4):160–163. pages 9

[26] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. The MIT Press, 2018. pages 2, 4

[27] Richard S. Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. 112(1):181–211. pages 8

[28] Keras Team. Simple. flexible. powerful. Accessed: 2021-01-23. pages 14

[29] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. pages 8

[30] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. page 9. pages 8

[31] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. 8(3):279–292. pages 7

[32] Christopher John Cornish Hellaby. Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge, 1989. pages 7

[33] Centrum Wiskunde. *DoubleQ-learning Hado van Hasselt Multi-agent and Adaptive Computation Group*. pages 8

[34] Chao Yu, Jiming Liu, and Shamim Nemati. Reinforcement learning in healthcare: A survey. pages 2

[35] Ziyao Zhang, Liang Ma, Kin K. Leung, Leandros Tassiulas, and Jeremy Tucker. Q-placement: Reinforcement-learning-based service placement in software-defined networks. *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018. pages 2

[36] Ziyao Zhang, Liang Ma, Konstantinos Poularakis, Kin K. Leung, and Lingfei Wu. Dq scheduler: Deep reinforcement learning based controller synchronization in distributed sdn. *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, 2019. pages 2

[37] Özgür Şimşek, Alicia P. Wolfe, and Andrew G. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22nd international conference on Machine learning*, ICML '05, pages 816–823. Association for Computing Machinery. pages 42

Appendix A

Performance graphs

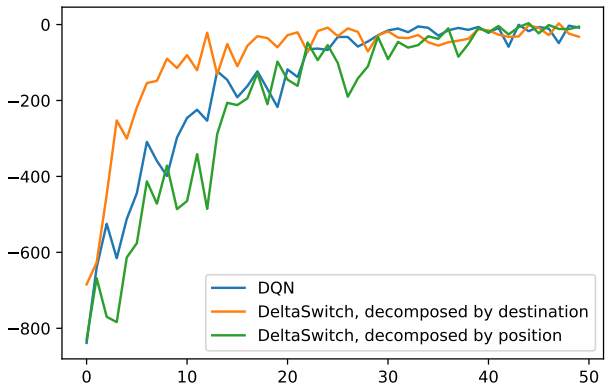


Figure A.1: Performance of DQN and 'DeltaSwitch' when the state is one-hot encoded.

Appendix B

Code

```
def stateDecomposition(st_table, thr):
  def joinGroups(i, j, d, groups):
    M = max(d[i], d[j])
    m = min(d[i], d[j])
    if d[i] != d[j]:
      groups[m] += groups[M]
      del groups[M]
      for ind, el in enumerate(d):
        if el > M: d[ind] -= 1
        if el == M: d[ind] = m

  n_states = st_table.shape[0]
  d = [i for i in range(n_states)]
  groups = [[i] for i in range(n_states)]

  #iterate through all the entries
  for i, j in np.ndindex(st_table.shape):
    if st_table[i][j] > thr: joinGroups(i, j, d, groups)

  return d, groups
```

Listing B.1: Function which is given the state-transition matrix in input and combines the thresholding of its values with the decomposition into separate MDPs, returning `d` which is a list indicating which sub-space each state belongs to (as an index) and `groups` which is a list of lists containing the states of each sub-space.

```
def stateDecompositionWithNGroups(st_table, target_n_groups):
  limitIterations = 50 # number of iterations after which the algorithm stops
  firstChangeFactor = 10.0 # factor by which the threshold is multiplied/divided
      after first iteration
  thr = 1.0 # initial threshold

  lowerBound = None
  upperBound = None

  for i in range(limitIterations):
    if lowerBound == None and upperBound != None: thr = upperBound /
        firstChangeFactor
    if lowerBound != None and upperBound == None: thr = lowerBound *
        firstChangeFactor
    if lowerBound != None and upperBound != None: thr = (upperBound + lowerBound)
         / 2.0
```

```
    stateGroups = stateDecomposition(st_table, thr)
    n_groups = len(stateGroups[1])
    if n_groups == target_n_groups:
      print("Target number achieved")
      break
    if n_groups < target_n_groups: lowerBound = thr
    if n_groups > target_n_groups: upperBound = thr
    print(thr)

  return stateGroups, thr
```

Listing B.2: Function that given the state-transition and a target number of subspaces performs the state-decomposition with different thresholds in order to obtain the correct number of sub-spaces.

```
class StateDecompositionDQNAgent:
  def __init__(self, state_size, action_size, subnetIndexes, subnets):
    self.state_size = state_size
    self.action_size = action_size
    self.gamma = 0.95    # discount rate
    self.epsilon = 1.0  # exploration rate
    self.epsilon_min = 0.01
    self.epsilon_decay = 0.999
    self.learning_rate = 0.001
    self.subnets = subnets
    self.n_subnets = len(stateGroups[1])
    self.subnetIndex = self._subnetIndex_init(subnetIndexes)
    self.memory = [deque(maxlen=50000) for i in range(len(self.subnets)+1)]
    self.models = self._build_models()

  def _subnetIndex_init(self, subnetIndexes):
    ind0s = subnetIndexes
    ind1s = [np.where(self.subnets[ind0s[i]] == i)[0][0] for i in range(self.
        state_size)]
    return list(zip(ind0s, ind1s))

  def _number_to_one_hot(self, n_classes, x):
    one_hot_vect = np.eye(n_classes)[x]
    reshaped = np.reshape(one_hot_vect, (1, n_classes))
    return reshaped

  def _one_hot_subnet(self, state):
    ind0, ind1 = self.subnetIndex[state]
    n_classes = self.subnets[ind0].shape[0]
    return self._number_to_one_hot(n_classes, ind1)

  def _all_zeros(self, subnet):
    l = len(subnet)
    return np.zeros((1,l))

  def _one_hot_joined(self, state):
    ind0, _ = self.subnetIndex[state]
    one_hot_inputs = []
    for i, subnet in enumerate(self.subnets):
      if i == ind0:
        one_hot_inputs.append(self._one_hot_subnet(state))
      else:
        one_hot_inputs.append(self._all_zeros(subnet))
    return np.concatenate(one_hot_inputs, axis=1)
```

```
def _build_subnet_model(self, subnet):
  subLen = len(subnet)
  inputs = Input(shape=(subLen,), name="input{}".format(subLen))
  x = Dense(50, activation='relu', name="subnet{}_dense_0".format(subLen))(
      inputs)
  x = Dense(50, activation='relu', name="subnet{}_dense_1".format(subLen))(x)
  x = Dense(50, activation='relu', name="subnet{}_dense_2".format(subLen))(x)
  out = Dense(self.action_size, activation='linear', name="subnet{}_dense_3".
      format(subLen))(x)
  model = Model(inputs=inputs, outputs=out, name="model{}".format(subLen))
  model.compile(loss='mse',
                optimizer=Adam(lr=self.learning_rate))
  return model

def _build_models(self):
  return [self._build_subnet_model(s) for s in self.subnets]

def join_models(self):
  def _joined_subnets(input):
    ind = 0
    outputs = []
    for i, subnet in enumerate(self.subnets):
      input_subnet = Lambda(lambda x: x[:,i:i+len(subnet)], name="lambda_{}".
          format(i))(input)
      return Lambda(lambda x: x[:,i:i+len(subnet)], name="lambda_{}".format(i))
          (input)
      outputs.append(self.models[i](input_subnet))
      i += len(subnet)
    return Concatenate(name="concatenate_int_outputs")(outputs)

  def _top_network(input):
    n_inputs = self.action_size * self.n_subnets
    x = Dense(24, activation='relu', input_shape=(None,n_inputs,), name="
        top_dense_0")(input)
    x = Dense(48, activation='relu', name="top_dense_1")(x)
    x = Dense(48, activation='relu', name="top_dense_2")(x)
    x = Dense(self.action_size, activation='linear', name="top_dense_3")(x)
    return x

  input = Input(shape=(self.state_size,), name="intermediate_input")
  int_outputs = _joined_subnets(input)
  output = _top_network(int_outputs)
  model = Model(inputs=input, outputs=output, name="joined_model")
  model.compile(loss='mse',
                optimizer=Adam(lr=self.learning_rate)) #maybe should have
                    separate lr for the top model
  self.models.append(model)

def remember_subnet(self, state, action, reward, next_state, done):
  ind0, _ = self.subnetIndex[state]
  next_ind0, _ = self.subnetIndex[next_state]
  if ind0 == next_ind0:
    self.memory[ind0].append((state, action, reward, next_state, done))
    return ind0
  self.memory[self.n_subnets].append((state, action, reward, next_state, done))
  return -1

def remember_joined(self, state, action, reward, next_state, done):
  self.memory[self.n_subnets].append((state, action, reward, next_state, done))
```

```
def act_subnet(self, state):# We implement the epsilon-greedy policy
  if np.random.rand() > self.epsilon:
    one_hot_state = self._one_hot_subnet(state)
    model = self.models[self.subnetIndex[state][0]]
    act_values = model.predict(one_hot_state)
    return np.argmax(act_values[0]) # returns action
  return random.randrange(self.action_size)

def exploit(self, state): # When we test the agent we dont want it to explore
    anymore, but to exploit what it has learnt
  act_values = self.models[self.n_subnets].predict(self._one_hot_joined(state))
  return np.argmax(act_values[0])

def act_joined(self, state):# We implement the epsilon-greedy policy
  if np.random.rand() > self.epsilon:
    return self.exploit(state) # returns action
  return random.randrange(self.action_size)


def replay(self, subnet_ind, batch_size, joined):
  if joined:
    index = self.n_subnets
  else:
    index = subnet_ind

  minibatch = random.sample(self.memory[index], batch_size)

  b = False
  for el in minibatch:
    if self.subnetIndex[el[0]][0] != subnet_ind or self.subnetIndex[el[3]][0]
        != subnet_ind:
      return True
  if b: print("WRONG")

  action_b = np.squeeze(np.array(list(map(lambda x: x[1], minibatch))))
  reward_b = np.squeeze(np.array(list(map(lambda x: x[2], minibatch))))
  done_b = np.squeeze(np.array(list(map(lambda x: x[4], minibatch))))

  ### Q-learning
  if joined:
    state_b_one_hot = np.squeeze(np.array(list(map(lambda x: self.
        _one_hot_joined(x[0]), minibatch))))
    next_state_b_one_hot = np.squeeze(np.array(list(map(lambda x: self.
        _one_hot_joined(x[3]), minibatch))))
  else:
    state_b_one_hot = np.squeeze(np.array(list(map(lambda x: self.
        _one_hot_subnet(x[0]), minibatch))))
    next_state_b_one_hot = np.squeeze(np.array(list(map(lambda x: self.
        _one_hot_subnet(x[3]), minibatch))))

  pred = self.models[index].predict(next_state_b_one_hot)
  target = (reward_b + self.gamma * np.amax(pred, 1))
  target[done_b==1] = reward_b[done_b==1]
  target_f = self.models[index].predict(state_b_one_hot)

  for k in range(target_f.shape[0]):
    target_f[k][action_b[k]] = target[k]
  self.models[index].train_on_batch(state_b_one_hot, target_f)
  if self.epsilon > self.epsilon_min:
```

```
        self.epsilon *= self.epsilon_decay

def load(self, name):
    self.model.load_weights(name)
def save(self, name):
    self.model.save_weights(name)
```

Listing B.3: Current version of my state-decomposition agent. In this version the states are fed in the neural networks as one-hot-encoded integers.