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

Comment this out if not needed.

Abstract

Your abstract

Contents

Chapter 1

Introduction

The introduction should set the scene and give a highlevel problem statement/specification, so that after reading the introduction the reader understands roughly what the problem is and what you intend to do about it. Is the idea to write software, or develop an algorithm, or produce hardware, or something else? You should then highlight and summarise the most interesting or important questions or problems that your project addresses, and the broader context in which those questions or problems are situated. Finally, you must briefly introduce the structure of report (what you will cover in which chapters and how these relate to each other). You don't need to go into any detail, the aim is to make sure the reader has an idea about what will be discussed and in what order

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 [1] 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 [2], a program that in 2016 beat the top-ranked Go player in the world. Other successful applications are in healthcare [3], robotics [4], autonomous driving [5] and natural language processing [6]. 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 [7] and the synchronisation of controllers in distributed SDNs [8].

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 the state-decomposition method proposed by my supervisor Prof. K. K. Leung. This 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 first identifies the sub-problems, then it trains separate RL agents for each of them. These are later combined to form a global agent that controls the whole system and takes into account their interactions.

While the high-level methodology is defined, the specifics of the algorithms are not. This project involves conducting research on the development of the state-decomposition method and to test it and benchmark it against mainstream algorithms to determine if it achieves improved performance.

1.3 Structure of the report

Provide info about structure

Chapter 2

Background

You should provide enough background to the reader for them to understand what the project is all about, and what is the relevant prior work. TJWC Final Year Projects 2 Examiners like to know that you have done the appropriate background research and it is important that you review either what has been done previously to tackle related problems, or perhaps what other products exist related to your deliverable. Clear references are important here, and much of this section will typically already have been written in your Interim Report. You may use feedback from that report to improve what you write in your Final Report, and should note that self-plagiarism between the two reports is not possible, so no citation is needed of your own earlier writing. - What does the reader need to know in order to understand the rest of the report? What problem are you solving? - Why is this problem interesting or worthwhile to solve? - Who cares if you solve it? - How does this relate to other work in this area? - What work does it build on? - For 'research-style' projects involving the design and analysis of specific algorithms there is a large amount of relevant background both of general theory, and very specific to the algorithm you investigate. Supervisors will help you to see what is most important here, but the general rule is that you must both provide overall context and note work close to what you do that influences your work or is in some way comparable to your work.

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" [1].

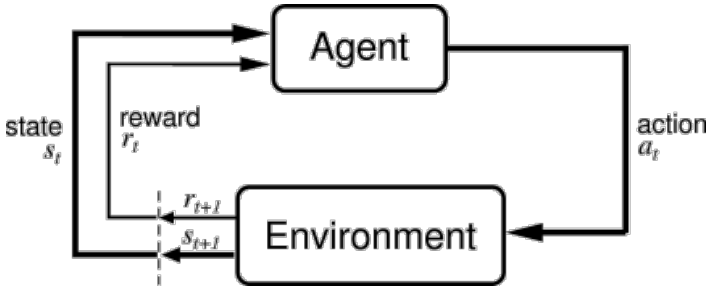


Figure 2.1: Interaction of the agent with the environment during time step t. Figure taken from [1]

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

[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 $S_{t+1} = S_t$

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 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.

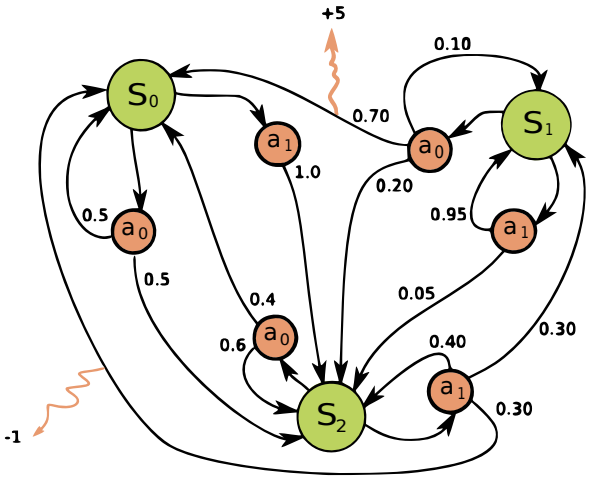


Figure 2.2: Graph representation of a Markov Decision Process. Note the rewards (zigzag arrows) that are released in certain transitions. [9]

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 [10]. One such method called *iterative policy evaluation*,

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

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

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

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

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 [11] 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 [12]. 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) [13] 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 [14]. 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

[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.

of the action values, which can lead to poor performance in some stochastic environments. Double Q-learning [15] 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 [16]. Another version of DQN that achieved improved performance in typical benchmarks such as playing Atari games is Dueling DQN [17], 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 [18].

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 [14] exploits the ability of deep learning to learn from high-dimensional data and to generalise

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

to solve this issue, though this is often not sufficient. [19] jointly learns state and action embeddings to improve generalisation. In World Models [20], 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 [21], 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 [22].

Hierarchical reinforcement learning [23] 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 [24], Options [25] and Hierarchies of Machines (HAM) [26].

Model-based reinforcement learning [27] has been receiving increasing attention lately. This class of algorithms is at the intersection 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 more efficiently learn from new samples. Dyna [28] 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 [2], 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. [20] 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

3.1 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. Separate RL agents can be trained for these separate sub-problems independently, thus treating them as separate problems. The separate agents are then combined at a later stage to take into account the interactions between the sub-problems.

The first step of the process is to identify the subsystems as 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 $\mathcal{S}$.

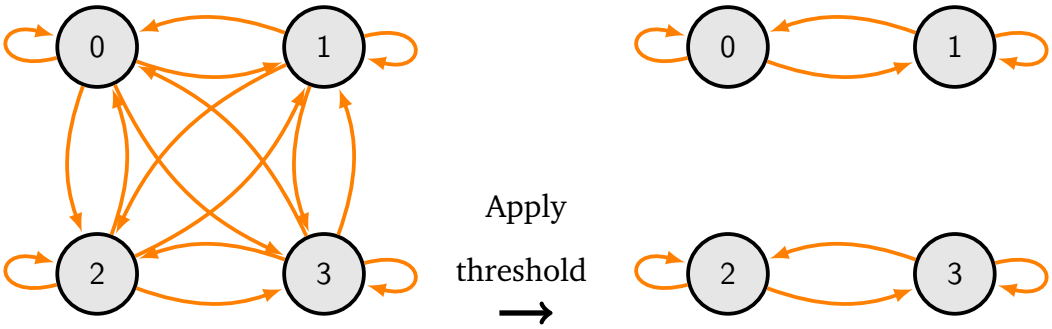


Figure 3.1: Applying a threshold to the state transition matrix and normalising the probabilities can cause the MDP to separate into separate MDPs as it reduces the number of possible transitions.

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 threshold ϵ 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 MDP could be formed by 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:

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

where the matrices $B_1, B_2, ..., B_n$ are the state-transition matrices of the newly formed separate MDPs.

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, for each of these sub-spaces, we train a separate DQN agent. 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[1], 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.

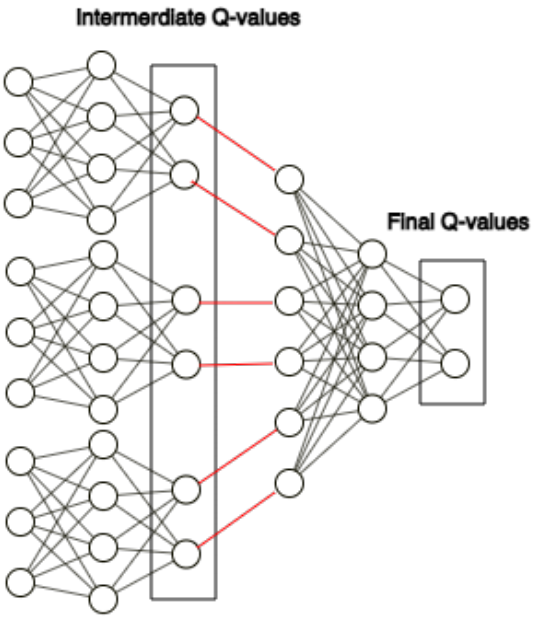


Figure 3.2: The NNs of the decomposed states' agents are merged as input of another NN to form a new bigger NN.

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

[1]Or earlier. The time when we should do this is to be researched in this project.

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[2]. 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.

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

3.2 Decomposing the state transition matrix

The decomposition into sub-spaces of states given the state-transition table is done using Algorithm-1. The Python code for this is given in the Appendix.

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] > ϵ then joinSubspacesContaining(i, j);
    end
end
```

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 ϵ at each iteration using binary search, until the MDP is decomposed in the correct number of sub-spaces. This is given in Algorithm-2 and the Python

code is in the Appendix.

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 $\leftarrow 10$;
$\epsilon \leftarrow 1.0$;

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

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

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

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

Chapter 4

Experimental method

The state-decomposition method is going to be implemented and tested in multiple varieties to assess the effect it has when applied to reinforcement learning algorithms.

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 (such as the OpenAI Gym toolkit previously described). Additionally, Python can be run on Colab notebooks.

Google Colaboratory Google 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.

[1]Tensor Processing Unit: a custom hardware accelerator developed by Google to accelerate the linear algebra operations that occur in machine learning. [29]

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.

Keras It was decided to implement the deep-learning models using the deep-learning API Keras [30]. This is a high-level interface that uses Tensorflow [31] for its back-end. 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” [32], which is currently the most commonly used toolkit to benchmark reinforcement learning algorithms. This toolkit offers a variety of different environments, ranging from classical control problems such as ”CartPole-v0” [33], 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.

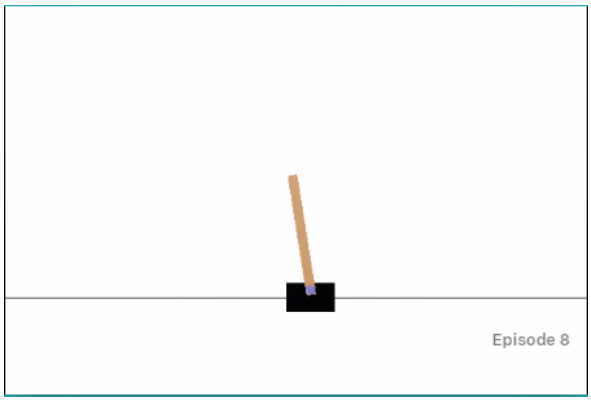


Figure 4.1: The CartPole-v1 environment is one of the most well-known environments of the OpenAI Gym library in which the goal is to balance the cartpole while keeping it within the boundaries of the screen by applying a leftwards or a rightwards force at each time-step. Screenshot of [33]

4.2 Obtaining 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.

4.3 The need for a suitable environment

The state-transition matrices and their decomposed versions were obtained for 4 different OpenAI Gym environments: ”Cartpole-v1”[2], ”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 is a more suitable candidate. The environment presents:

- 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.

[2]This environment has continuous states which are discretised using a linear quantiser to apply the decomposition algorithm.

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 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.

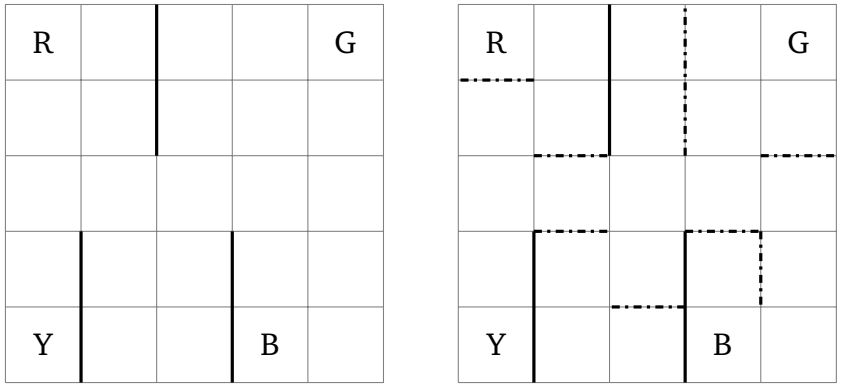
These problems in finding a suitable environment suggest that the best solution would be to create a custom environment with the characteristics that would allow better testing of the method. The ”Taxi-v3” environment is composed of 4 independent MDPs. Modifying its transition-probabilities so that interaction between them is possible[3], would make this environment suitable for testing the decomposition algorithm. For this purpose, I propose 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

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

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.



(a) ”Taxi” **(b)** ”TaxiTraps”

Figure 4.2: On the left, the original ”Taxi” environment. On the right, the ”TaxiTraps” environment.

4.4 Implementation of the DQN agents

Some initial testing of the state decomposition method has been conducted in the ”Taxi-v3” environment. Here, the state-decomposition method modified to skip the second stage of joining the networks together showed equivalent performance as a single DQN agent. The equivalence is to be expected since in this environment the sub-problems are completely independent and it does not prove anything about the algorithm in general. One issue of the testing conducted until now is the format in which the states were fed into the Neural Networks. The OpenAI Gym library repre-

sents discrete states and actions as single integers. Such a representation is optimal when using tabular Q-learning, as the representations of states and actions simply act as indexes and don't need to be meaningful. However, when using a Q-function approximator, such a meaningless representation impedes the generalisation properties in unseen states that make DQN so useful. Representing the states as one-hot encoded versions of the original integer has also been tested. Though this could allow using shallower neural networks as the decoding of the states is simpler, this is still not a meaningful representation of the states. The current state of the code of the state-decomposition agent is given in the Appendix. To take full advantage of the generalisation to unseen states property of DQN, the format in which states are input to the agents will be modified to be a meaningful representation. This could be in a vector format, where each element of the vector represents a different aspect, such as position, destination, etc.

4.5 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 hat 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 the following section.

The chosen benchmark agent is a basic DQN agent. The value updates occur with discount factor $\gamma = 0.95$ WHY??? (I think even though the episode are finite so could use gamma=1, using a lower gamma enhaces the stability of the algorithm). The agents 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 (RANDOM OR ONE OF THE OTHERS? NOt big difference anyways). 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.

The neural network that approximates the Q-values is fully connected and it has 4 layers. The input state is fed as an integer to an encoding layer[4], which then feeds to the first fully-connected layer, consisting of 512 neurons with "ReLU"[5] 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.

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

[5]"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

Testing

Describe your Test Plan – how the program or system was verified. Put the actual test results in an Appendix if they are repetitive but relevant. Detailed test data may be omitted from the report if not relevant, however an accurate summary of tests should be included in the Report itself. Sometimes non-working designs are described in project reports as though they work, when in reality they don't, or only partially work. Therefore a precise description of what works and how this has been established is important. Examiners may try to compile, use, or test deliverables themselves (even after your report is submitted), and your report should accurately reflect the state of the project. This section is normally useful for software or hardware deliverables and less relevant in analytical projects.

Chapter 6

Results

This covers an area different from the 'Testing' chapter, and relates to the understanding and analysis of the algorithm, program, or hardware behaviour. Where the deliverable is a product with easily quantified performance then the previous Chapter shows how functional correctness is established, while this Chapter shows qualitatively or quantitatively how well the product works. The list below shows typical content, not all of this will be applicable to all projects. - An empirical relationship between parameters and results may be investigated, typically through the use of appropriate graphs. - Where theory predicts aspects of performance the results can be compared TJWC Final Year Projects 4 with expectation and differences noted and (if possible) explained. - Semi-empirical models of behaviour may be developed, justified, and tested. - The types of experiments/simulations that were carried out should be described. Why were certain experiments carried out but not others? What were the important parameters in the simulation and how did they affect the results? If there are very many graphs and tables associated with this chapter they may be put in the Appendix, but it is generally better to keep these close to the text they illustrate, as this is easier for the reader.

6.1 Needing to use one-hot encoding

6.2 Initial results with Taxi2 and original state decomposition algorithm

6.3 Effect of different encoding methods

6.4 Effect of different number of parameters

6.5 Effect of using reduced encoding (taking advantage of state decomposition for smaller input vector)

6.6 Performance gain of DeltaSwitch

6.7 Samples plots

6.8 Effect of wall / no wall

6.9 Effect of decomposing by destination or by position

6.10 Techniques to learn switch and relationship with transfer learning + fair comparison

Chapter 7

Evaluation Plan

The main objective of this project is to evaluate the performance of the proposed state-decomposition algorithm. To do so, the algorithm will be tested in different varieties in the TaxiTraps environment described in Section-4.3. The possible varieties are:

- In the original state-decomposition algorithm it is assumed that the NNs of the sub-spaces' agents keep training after they are joined together in the second stage of the training as they are part of the bigger newly formed neural network of the global agent. Alternatively, we could freeze the weights of the sub-spaces' NNs (trained in the 1st stage) during the second stage of training.

- Once the NNs are joined in the 2nd stage, the training could be done using all transitions, only those that start and end in different sub-space[1] or start with only transitions between different sub-spaces[2] and then as the global agent experiences more transitions use all possible transitions.

- The time at which the agents of the sub-spaces should be joined together to form the global agent is not defined. Different strategies can be compared, such

[1] To take into account the transitions between different sub-spaces.

[2] At this stage of training we have many of these transitions that were saved but ignored in the first stage of training.

as waiting for the sub-agents' action-values to converge or defining different minimum rates of change of the average episode rewards going below which would start the second stage of training.

The main aspect of these algorithms that will be evaluated is the sample efficiency, which is how efficiently the algorithm uses the samples that it experiences to learn a policy. This affects how quickly the algorithm can learn an optimal policy and it can be measured by plotting the total reward per episode vs the number of training steps. It is also important to consider the fact that the learning process is stochastic and the performance varies between trials, hence we normally express the reward per episode using a confidence bound.

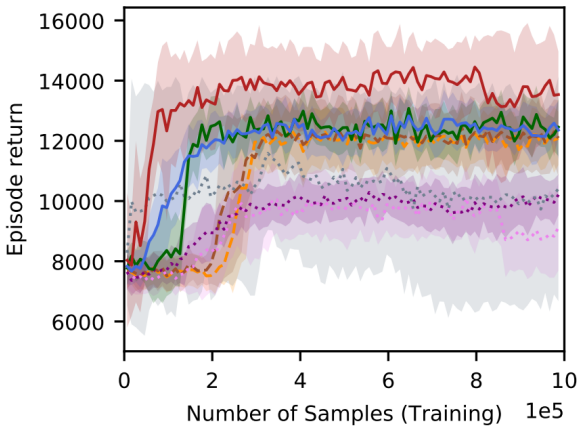


Figure 7.1: Example of reward per episode vs number of training steps. Note the confidence bands to express the uncertainty in the results. Figure taken from [19]

All these variations of the state-decomposition method will be compared with a standard DQN agent. There are many scenarios where the system is known to be formed by almost independent sub-systems corresponding to almost independent sub-problems. In such a scenario, the states could be grouped into separate subspaces without even estimating the state-transition matrix by just leveraging the knowledge of the characteristics of the system. Thus, we shall compare the performance of the state-decomposition method when the state-transition matrix is given

(or sub-spaces can be guessed) at the start and when it has to be estimated from samples. In the latter case, we can estimate the model of the environment is we also consider the rewards of the samples. Thus we shall also compare the performance of the state-decomposition algorithm with a method that simply approximates the model of the environment from samples and then finds the optimal policy using planning algorithms.

THINGS TO CARE ABOUT:

When I use the deltaswitch, how the weights updated for NN0 when NN1 is active and vice-versa. https://www.gatsby.ucl.ac.uk/ dayan/papers/cjch.pdf https://arxiv.org/pdf/1901.0

relationship with drpout gains might be due to effects of double q-learning rather than decomposition resource allocation / job allocation alibaba datasetF

Chapter 8

Evaluation

This Chapter (or possibly section of the conclusions) is distinct from your results. It must contain your critical evaluation of your work as compared to previous analysis, algorithms, products, and when related to your original objectives. To what extent have your original objectives been fulfilled? If they have changed, what is your rationale for this? What are the advantages, disadvantages of your approach compared with related work? How does the scope of your work differ from related work? Examiners expect your project report to show evidence of your ability to think as an engineer, and that includes the ability to critically reflect on your own work and evaluate its significance. Material here will compare project outcomes with initial objectives and requirements captured. Usually your Interim Report will contain these. Where these have changed significantly over the course of the project this should be explained and reasons given. This section should not require examiners to read your Interim Report, and will not reference it. Changes between final and initial objectives should be explained in a self-contained manner. Note that here you will reference and summarise, rather than repeat, your description of Requirements Capture earlier in the Final Report.

Chapter 9

Conclusions and Further Work

How successful have you been? What have you achieved? How could the work be taken further given more time (perhaps by another student next year)? It is important here to identify positively what is worthwhile in your work. At the same time, honesty, and a clear description of the limits of your work, is equally important. It is often most appropriate to describe work you did not have time to complete as further work. Your readers will not be clear where, in your long report, are your most significant achievements. In the conclusions you must summarise this, referring as necessary to other sections for more detail. - What design choices did you have along the way, and why did you make the choices you made? - What was the most difficult and/or clever part of the project? - Why was it difficult? - How did you overcome the difficulties? - Did you discover or invent anything novel? - What did you learn? Note that "difficult" does not necessarily mean the thing that took you the longest amount of time. Note also that the conclusions must concisely summarise this material, and refer to other sections for the details.

9.1 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.

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-9.1, 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.

Offline training The samples obtained from the pre-training interaction required to obtain the state-transition matrix could be saved and later added to the experience replay memories of the DQN agents. Offline pre-training could also be conducted using these samples as this could improve sample efficiency. Similarly, transition samples between different sub-spaces are not used in the first stage of training. These could be used to conduct offline pre-training of the merged network.

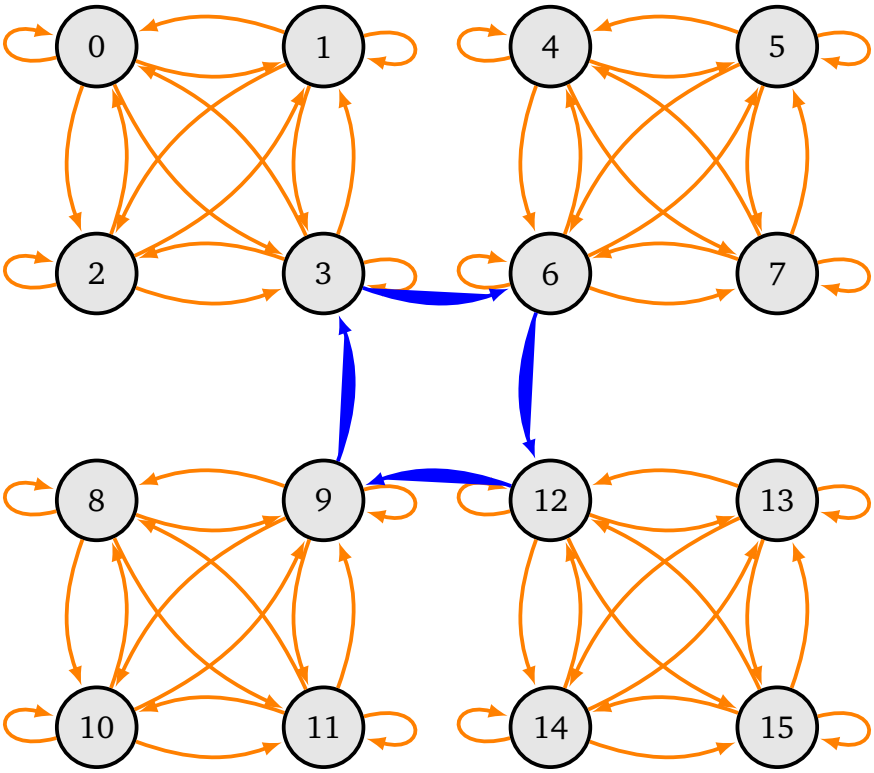


Figure 9.1: MDP composed of smaller sub-spaces of states connected by few but probable transitions.

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 [34] and [35], 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-9.1.

9.2 Application to real-life scenarios

Resource allocation problem, 2 data centres, allocate job to 2nd data-centre Driving taxi Alibaba thing

Chapter 10

Ethical, Legal, and Safety Plan

Following a brief consultation with my project supervisor, I can state that this project does not involve any kind of ethical, legal, or safety issues.

Ethical Issues The project does not involve experimentation with humans or animals and it is not going to cause any physical or psychological harm to any participants.

Legal Issues The project doesn't involve the participation of people in surveys or any kind of collection of sensitive data, meaning that privacy is not an issue. This is an open research project and it is not affected by any copyrights or confidentiality agreements. This work is also not affected by any existing patents.

Safety Issues The project work is extremely unlikely to cause any physical harm as it doesn't involve the operation of dangerous equipment involving any high voltages, chemicals, flammable materials, biological hazards, etc.

Bibliography

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction*. The MIT Press, 2018. pages 2, 6

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, pp. 354–359, Oct. 2017. Number: 7676 Publisher: Nature Publishing Group. pages 3, 14

[3] C. Yu, J. Liu, and S. Nemati, "Reinforcement Learning in Healthcare: A Survey," *arXiv:1908.08796 [cs]*, Apr. 2020. arXiv: 1908.08796. pages 3

[4] "Google AI Blog: Scalable Deep Reinforcement Learning for Robotic Manipulation." pages 3

[5] R. B Kiran, I. Sobh, T. Victor, P. Mannion, A. A. Al Sallab, S. Yogamani, and P. Pérez, "Deep reinforcement learning for autonomous driving: A survey," *arXiv.org*, Feb 2020. pages 3

[6] R. Paulus, C. Xiong, and R. Socher, "A deep reinforced model for abstractive summarization," November 2017. pages 3

[7] Z. Zhang, L. Ma, K. K. Leung, L. Tassiulas, and J. Tucker, "Q-placement: Reinforcement-learning-based service placement in software-defined networks," *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018. pages 3

[8] Z. Zhang, L. Ma, K. Poularakis, K. K. Leung, and L. Wu, "Dq scheduler: Deep reinforcement learning based controller synchronization in distributed sdn," *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, 2019. pages 3

[9] "Markov decision process," Jan. 2021. Page Version ID: 1002665350. pages 8

[10] R. Bellman, "Dynamic Programming," *Science*, vol. 153, pp. 34–37, July 1966. pages 9

[11] C. J. C. H. Watkins, *Learning from delayed rewards*. PhD thesis, University of Cambridge, 1989. pages 11

[12] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, May 1992. pages 12

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015. pages 12

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *arXiv e-prints*, p. arXiv:1312.5602, Dec. 2013. pages 12, 13

[15] C. Wiskunde, *DoubleQ-learning Hado van Hasselt Multi-agent and Adaptive Computation Group*. pages 13

[16] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," *arXiv:1509.06461 [cs]*, Dec. 2015. arXiv: 1509.06461. pages 13

[17] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling Network Architectures for Deep Reinforcement Learning," p. 9. pages 13

[18] C. Gaskett, D. Wettergreen, and A. Zelinsky, "Q-Learning in Continuous State and Action Spaces," in *Advanced Topics in Artificial Intelligence* (G. Goos, J. Hartmanis, J. van Leeuwen, and N. Foo, eds.), vol. 1747, pp. 417–428, Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. Series Title: Lecture Notes in Computer Science. pages 13

[19] P. J. Pritz, L. Ma, and K. K. Leung, "Jointly-Trained State-Action Embedding for Efficient Reinforcement Learning," *arXiv e-prints*, p. arXiv:2010.04444, Oct. 2020. pages 14, 34

[20] D. Ha and J. Schmidhuber, "World Models," *arXiv e-prints*, p. arXiv:1803.10122, Mar. 2018. pages 14, 15

[21] T. L. Dean, R. Givan, and S. Leach, "Model Reduction Techniques for Computing Approximately Optimal Solutions for Markov Decision Processes," *arXiv e-prints*, p. arXiv:1302.1533, Feb. 2013. pages 14

[22] M. Asadi and M. Huber, "State space reduction for hierarchical reinforcement learning," 2004. pages 14

[23] A. G. Barto and S. Mahadevan, "Recent Advances in Hierarchical Reinforcement Learning," *Discrete Event Dynamic Systems*, vol. 13, pp. 41–77, Jan. 2003. pages 14

[24] T. G. Dietterich, "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition," *arXiv:cs/9905014*, May 1999. arXiv: cs/9905014. pages 14

[25] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, vol. 112, pp. 181–211, Aug. 1999. pages 14

[26] R. Parr and S. Russell, "Reinforcement Learning with Hierarchies of Machines," p. 7. pages 14

[27] T. M. Moerland, J. Broekens, and C. M. Jonker, "Model-based Reinforcement Learning: A Survey," *arXiv:2006.16712 [cs, stat]*, July 2020. arXiv: 2006.16712. pages 14

[28] R. S. Sutton, "Dyna, an integrated architecture for learning, planning, and reacting," *ACM SIGART Bulletin*, vol. 2, pp. 160–163, July 1991. pages 14

[29] "Cloud Tensor Processing Units (TPUs)." pages 22

[30] K. Team, "Simple. flexible. powerful.." Accessed: 2021-01-23. pages 23

[31] "TensorFlow." pages 23

[32] OpenAI, "Gym: A toolkit for developing and comparing reinforcement learning algorithms." pages 23

[33] "OpenAI Gym: the CartPole-v0 environment." pages 23, 24

[34] O. Şimşek, A. P. Wolfe, and A. G. Barto, "Identifying useful subgoals in reinforcement learning by local graph partitioning," in *Proceedings of the 22nd international conference on Machine learning*, ICML '05, (New York, NY, USA), pp. 816–823, Association for Computing Machinery, Aug. 2005. pages 39

[35] I. Menache, S. Mannor, and N. Shimkin, "Q-Cut—Dynamic Discovery of Sub-goals in Reinforcement Learning," in *Machine Learning: ECML 2002* (T. Elomaa, H. Mannila, and H. Toivonen, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 295–306, Springer, 2002. pages 39

Appendix A

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