VRIJE
UNIVERSITEIT
BRUSSEL

Graduation thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Applied Science and Engineering: Computer Science

KNOWLEDGE TRANSFER IN DEEP REINFORCEMENT LEARNING

Arno Moonens

Academic Year: 2016-2017

Promotor: Prof. Dr. Peter Vrancx

Science and Bio-Engineering Sciences

VRIJE
UNIVERSITEIT
BRUSSEL

Proefschrift ingediend met het oog op het behalen van de graad van Master of Science in Applied Science and Engineering: Computer Science

KENNISOVERDRACHT BIJ DEEP REINFORCEMENT LEARNING

Arno Moonens

Academiejaar: 2016-2017

Promotor: Prof. Dr. Peter Vrancx

Wetenschappen en Bio-ingenieurswetenschappen

Abstract

Deep reinforcement learning allows for learning how to perform a task using high-dimensional input such as images by trial-and-error. However, it is sometimes necessary to learn variations of a task, for which certain knowledge can be transferred. In this thesis, we learn multiple variations of a task in parallel using both shared knowledge and task-specific knowledge. This knowledge is then transferred to a new task. In experiments, we saw that learning first in parallel on a set of source tasks significantly improves performance on a new task compared to not learning on source tasks or only using one. We also found that it is beneficial to start learning on a new task using task-specific knowledge of a source task.

Acknowledgments

I would like to thank my promotor Prof. Dr. Peter Vrancx for always giving useful feedback and helping me when I was stuck. This thesis would not have been possible without his expertise and the inspiration he offered. I would also like to thank my friends and family for always supporting me in this and all other endeavors.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

Reinforcement learning is a branch from the machine learning field where we we learn from an environment by interacting with it. The reinforcement learning algorithm selects an action, executes it and can receive a reward. It can then change its way of selecting actions in order to get a higher reward, which is possibly received later on.
The environment in which the agent acts can be for example a game like *Pong*, a smart home controlling the room temperature or a self-driving car.

Determining which action to take can be done by using an artificial neural network. It is inspired by the brain of a human and consists of interconnected elements called units. The network receives an input as a vector of numerical values. These are then propagated to layers of units and results in one or more output units.
In the context of reinforcement learning, these input units can be the current state of the environment. The output units can represent the action that has to be taken. The learning process can then involve changing the strength of connections between units and as such influencing the output values.

For high-dimensional inputs, different kinds of artificial neural networks must be used to be able to process these inputs. Techniques involving these networks are called deep learning methods. For example, an image can be used as input to the network. These typically include several thousands of pixels of each a certain color. Convolutional neural networks are able to detect patterns in the images using several layers of filters. Another deep learning network is the recurrent neural network, which is able to process a sequence of data such as a video or text.

By combining deep learning and reinforcement learning, it is possible to learn in an environment that has a high-dimensional input space. This is called deep reinforcement learning. For example, the agent can learn how to play *Pong* just using an image of the screen, just like a human.

An environment in which is learned is roughly defined by the possible states in which

it can be, which actions can be taken and in which state one ends up in when taking an action being in a certain state. These however can be changed such that the environment is easier or harder to learn. In an environment where a self driving car must be controlled for example, the amount of obstacles may vary or the weather conditions may change.
Although these changes may require different capabilities of the agent, some knowledge may still be useful. It can thus be beneficial for the agent to transfer the already learned knowledge from the initial situation, called *source task* to the agent learning in the new situation, called the *target task*. This domain is called transfer learning. One use of this is for example in cases where it is too expensive or time consuming to learn in the real world. Instead, one can first learn in a simulation and then transfer the knowledge to use and fine tune it in the real world, saving time and money.
It is necessary to know from which source tasks to transfer knowledge and which knowledge to transfer. For this, we need to know how the tasks are related and possibly how an agent can interpret and act using the new state space and action space.

In this thesis, we investigate the use of transfer learning in reinforcement learning using artificial neural networks. First, we will introduce the reader to the concepts of reinforcement learning, artificial neural networks, deep learning, deep reinforcement learning and transfer learning. Afterwards, we will discuss related work in the field of transfer learning applied to reinforcement learning and deep reinforcement learning. We will then explain the algorithm that was used in experiments and how the experiments were conducted. Last, the results of the experiments will be discussed.

Chapter 2

Artificial neural networks

Artificial neural networks are models that can approximate discrete-valued, real-valued and vector-valued functions. They are composed of interconnected units that can activate other units using connections of varying strength. Artificial neural networks are loosely inspired by biological neural networks, where these units are called neurons, which are connected by axons.
The higher the strength, also called weight, of connections between units, the more influence the unit has on the next one. These strengths are numerical values that can be manually defined. However, thanks to a method called backpropagation, weights can be learned systematically. The combination of artificial neural networks and backpropagation led to successful applications, for example to recognize handwritten characters (LeCun et al., 1989), for face recognition (Cottrell, 1990) and to recognize spoken words (Lang, Waibel, & Hinton, 1990). However, early applications also include reinforcement learning, for example to learn to balance a pendulum (Anderson, 1989) or to play the game *Backgammon* (Tesauro, 1992).

2.1 Basics

As said, artificial neural networks are made up of units. These units can be grouped into layers, where the values of each layer are propagated to the next layer depending on the weights of their connections. These weights define the model. Input units receive external information and provide information to other units. Output units are the opposite and receive information from the network itself and provide information externally. Hidden units both receive information from and provide information to units inside the network. A simple artificial neural network is visualized in Figure 2.1.

As a non-input unit gets information from multiple units, these need to be combined to determine if the unit can be activated or not. This is done by applying a linear or non-linear function, called an activation function, to the weighted sum of the connected units.

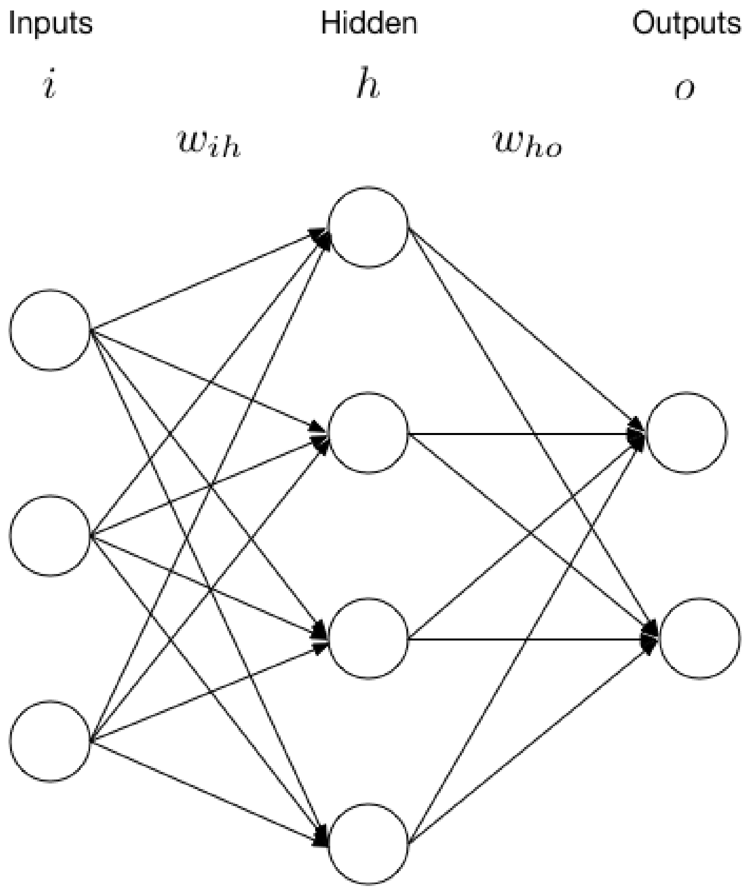

Figure 2.1: An artificial neural network with 3 input units, 4 hidden units and 2 output units. w_{ih} and w_{ho} are the weights for connections between respectively the input units and hidden units and between the hidden units and output units.

2.2 Activation functions

Formally, an activation function ϕ will compute for unit j with inputs from layer x:

$$o = \phi\big(\sum_i^n w_{ij} x_i\big) \tag{2.2.1}$$

Or in vector format:

$$o = \phi(\overrightarrow{w} \cdot \overrightarrow{x}) \tag{2.2.2}$$

From hereon we will denote the weighted sum as $z \equiv \sum_i^n w_{ij} x_i \equiv \overrightarrow{w} \cdot \overrightarrow{x}$.

2.2.1 Perceptron

A perceptron unit, defined by Rosenblatt, 1958, gives as output either -1 or $+1$ depending on the linear combination of the input and the weights:

$$o(\overrightarrow{x}) = \begin{cases} +1 & \text{if } \overrightarrow{w} \cdot \overrightarrow{x} > 0 \\ -1 & \text{otherwise} \end{cases} \tag{2.2.3}$$

It can be seen as a hyperplane where the output is -1 or $+1$ depending on which side the input lies. Using -1 as `false` and $+1$ as `true`, it is also possible to represent boolean functions such as *AND*, *OR*, *NAND* and *NOR*. However, some boolean functions, such as *XOR* cannot be represented by a single perceptron (Mitchell, 1997).

A schematic using the perceptron unit is shown in Figure 2.2.

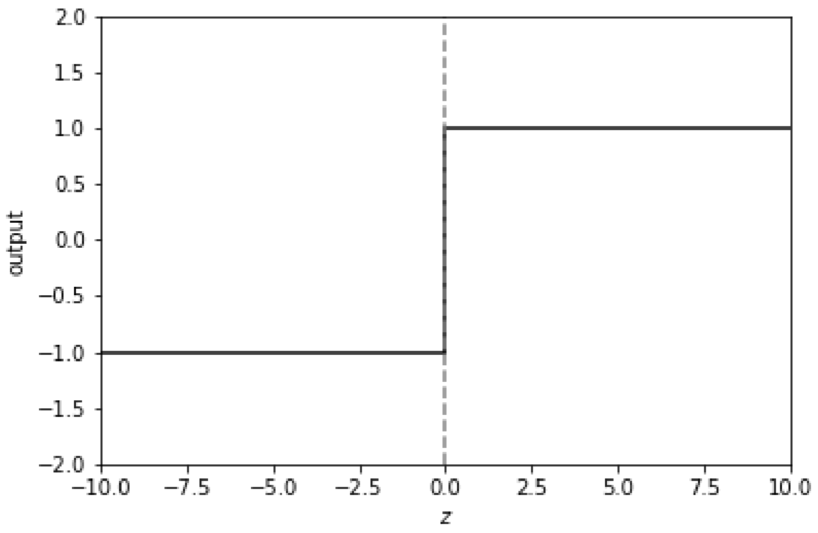

Figure 2.2: The perceptron unit. Source: Demant, Garnica, and Streicher-Abel, 2013.

2.2.2 Sigmoid

A disadvantage of the perceptron unit is that it is not differentiable in the whole domain (specifically at $x = 0$). Why this is a problem will become clear in Section 2.3.
To solve this problem, a sigmoid function can be used. A sigmoid function is a differentiable function that is monotonically increasing and approaches an asymptote for $x \to \pm\infty$ (LeCun, Bottou, Orr, & Müller, 2012). As a result, these can still separate the input space in 2 parts.

In the artificial neural networks domain, the *sigmoid function* generally refers to a variation of the logistic function and is denoted by $\sigma(x)$. Its formula is:

$$\begin{aligned} f(z) &= \sigma(z) \\ &= \frac{1}{1+e^{-z}} \end{aligned} \tag{2.2.4}$$

The behavior of this function is visualized in Figure 2.3.

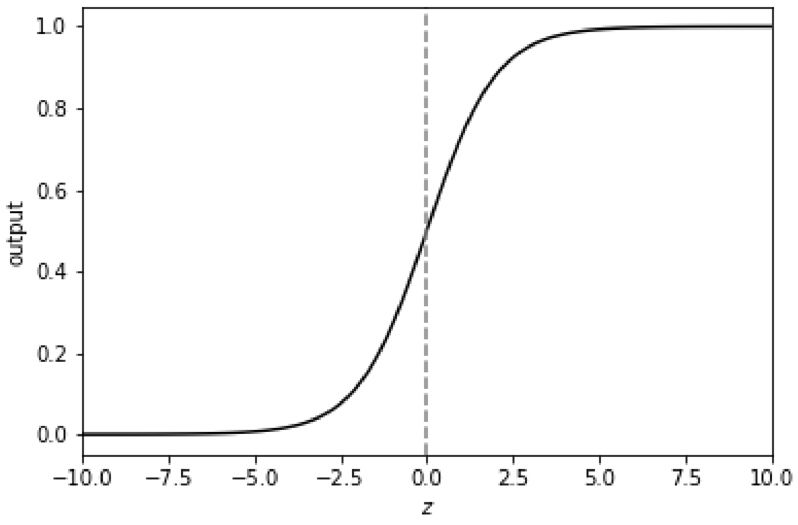

Figure 2.3: The sigmoid function.

2.2.3 Hyperbolic tangent

Another popular function and another kind of sigmoid function is the hyperbolic tangent function, also called *tanh*. While the output of the sigmoid function ranges between 0 and +1, here the output ranges between −1 and +1, just like the range (and the only possible values) of the perceptron. The formula of the hyperbolic tangent function, which can also be written in terms of the sigmoid function, is:

$$\begin{aligned} f(z) &= tanh(z) \\ &= 2\sigma(2z) - 1 \\ &= \frac{e^{\frac{z}{2}} - e^{-\frac{z}{2}}}{e^{\frac{z}{2}} + e^{-\frac{z}{2}}} \end{aligned} \tag{2.2.5}$$

This function is visualized in Figure 2.4.

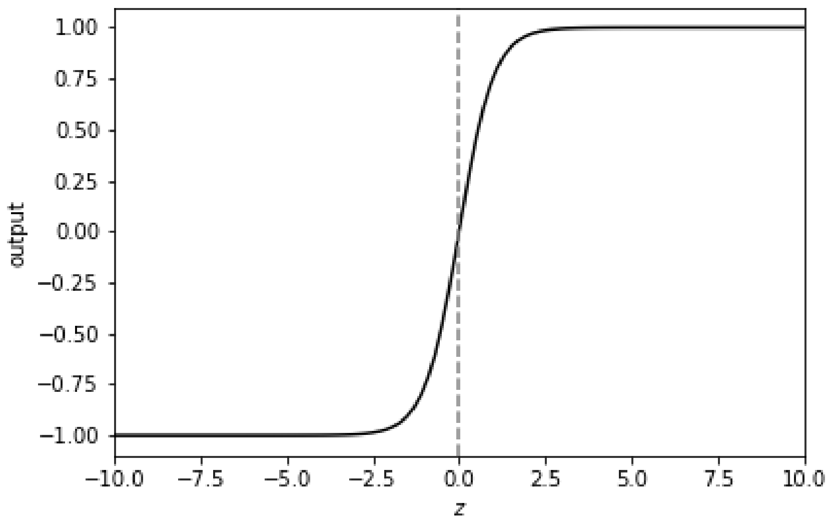

Figure 2.4: Hyperbolic tangent function.

2.2.4 Rectified Linear Unit

A rectified linear unit, also called a ReLU, is another popular and more recent activation function. It was first defined by Nair and Hinton, 2010. The output is the identity function if $z \geq 0$ and 0 otherwise:

$$f(z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.2.6}$$

As can be seen, this function requires less computations than the sigmoid and hyperbolic tangent function. However, the derivative for $z < 0$ is always 0. Why this can be a problem is also explained in Section 2.3.
The ReLU function is shown in Figure 2.5.

To solve the problem of the derivative being zero at $z < 0$, the Leaky ReLU was invented (Maas, Hannun, & Ng, 2013). Here, instead of the output being zero for $z < 0$, the output has a small slope, defined by a constant α:

$$f(z) = \begin{cases} z & \text{if } z \geq 0 \\ \alpha z & \text{otherwise} \end{cases} \tag{2.2.7}$$

As can be seen, the derivative of this function is always non-zero.
An example of a Leaky ReLU is shown in Figure 2.6.

A variation of a Leaky ReLU, called Parametric Rectified Unit, also has a small slope for $x < 0$, but with an α that can be learned (He, Zhang, Ren, & Sun, 2015). Other

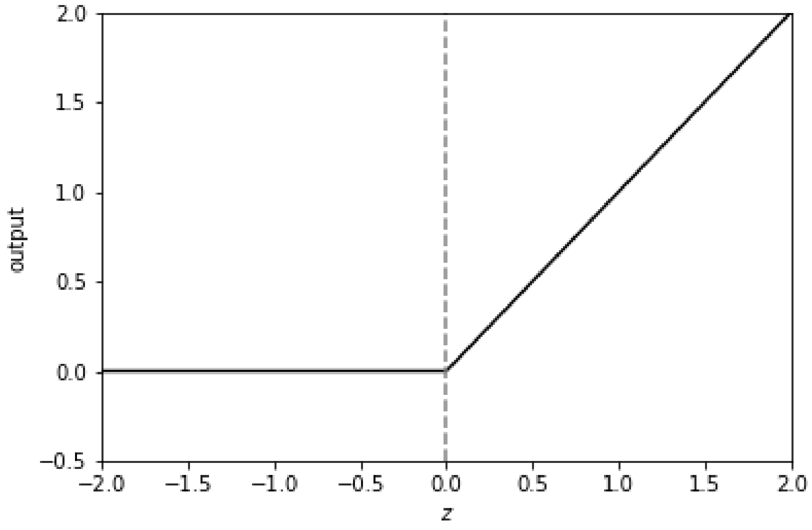

Figure 2.5: Rectified Linear Unit function.

variants use a random slope for values below zero (Xu, Wang, Chen, & Li, 2015), add noise a ReLU (Nair & Hinton, 2010) or use an exponential function for values below zero (Clevert, Unterthiner, & Hochreiter, 2015):

$$f(z) = \begin{cases} z & \text{if } z \geq 0 \\ \alpha(e^z - 1) & \text{otherwise} \end{cases} \tag{2.2.8}$$

2.2.5 Softmax

The softmax activation function is different from the previous activation functions that were described because here the output of one neuron depends on other neurons of the same layer:

$$f(z_i) = \frac{e^{z_i}}{\sum_j e^z_j} \tag{2.2.9}$$

Where z_i is the weighted sum for the current output to be computed and z_j is the weighted sum of another neuron in the same layer.
As we can see, the softmax function normalizes the outputs and results in outputs ranging between 0 and 1. Because of this, we can use the outputs as if they are probabilities. This is useful in a task where an action needs to be taken probabilistically based on the output of the artificial neural network.

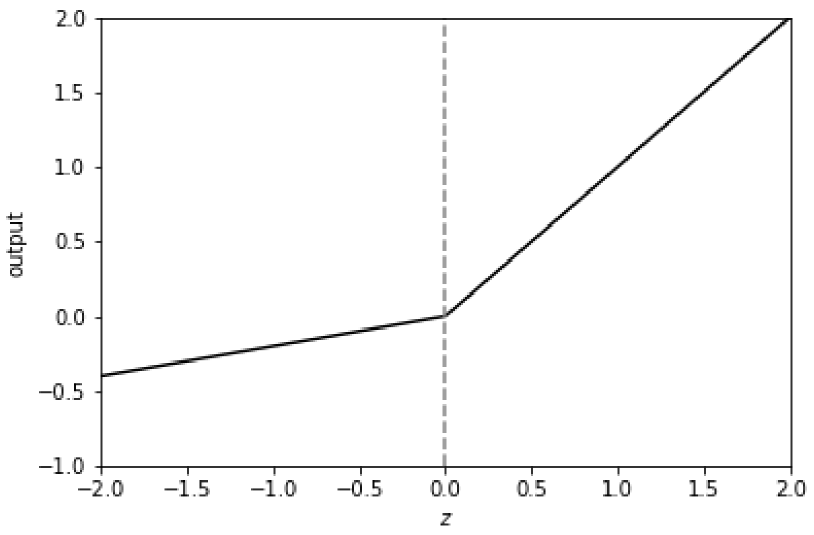

Figure 2.6: Leaky ReLU function with $\alpha = 0.2$.

2.3 Gradient descent and backpropagation

If we choose the weights of our network correctly, we might succeed in approximating a function. However, it is not always possible to separate the input space and have the right outputs using our artificial neural network. In that case, we might want the best possible solution, e.g. one that has the least amount of errors in producing outputs w.r.t. the correct outputs, also called the target outputs. Furthermore, choosing the weights manually can be a tedious process.
For these reasons, we use an algorithm called called backpropagation, which uses a variation of gradient descent. We will first discuss gradient descent as this provides the basis of the backpropagation algorithm.

2.3.1 Gradient descent

Gradient descent is a technique used to find a local minimum of a function using the gradient of that function with respect to its parameters. These parameters lie in a weight vector space, also called hypothesis space.

In our case, we want to approximate a function. As such, we want to minimize the differences between our outputs and the outputs of the function, also called the training examples. This difference, called the training error is also needed to update the weights. It depends on the learning algorithm (in our case an artificial neural network), its parameters and the training examples. However, here we assume that the learning algorithm and the training examples are fixed while learning.

A common measure for the training error, the mean squared error (MSE), is defined as:

$$E(\overrightarrow{w}) \equiv \frac{1}{N}\sum_{i=0}^{N}(y_i - f(x_i;\overrightarrow{w}))^2 \tag{2.3.1}$$

Where N is the number of training examples, x_i is the input of a training example, y_i is the target output of the training example and f is our learning algorithm, which depends on the weight vector $\overrightarrow{w}$.

Because we now know in which way the training error and the weight vector are related, we can compute the derivative of the training error E with respect to each component of $\overrightarrow{w}$. This called the gradient and is denoted as $\nabla E(\overrightarrow{w})$:

$$\nabla E(\overrightarrow{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n}\right] \tag{2.3.2}$$

Where n is the number of components in $\overrightarrow{w}$. Note that $\nabla E(\overrightarrow{w})$ is also a vector. It defines how to change the weight components in order to get the steepest increase in the training error E. Thus, if we negate the gradient, we get the steepest decrease in E. We can then update the weight vector as such:

$$\overrightarrow{w} \leftarrow \overrightarrow{w} - \eta\nabla E(\overrightarrow{w}) \tag{2.3.3}$$

Where η is a positive value called the learning rate, which influences how big the changes to the weight vector are.

The goal is to set the weights to such values so that the error is minimized. This is called a global minimum. No other combinations of weight values can lead to a lower error than those of the global minima.
However, it is possible that, given a combination of weight values, any change in those values does not influence the error, in which case the gradient for every weight component is 0. Thus, with the current algorithm, there is no incentive to change the weights. This is the case when every set of weights in the neighborhood of the current ones leads to a higher error. This "position" in the weight vector space is called a local minimum. Although the algorithm cannot "see" immediate improvements, it is possible that another combination of weight values leads to a lower error. This is analogous to being in a valley in the mountains, where the valley after a mountain (and thus not visible) may lie lower. This is also visualized in Figure 2.7.

For artificial neural networks without a hidden layer, such as a single perceptron unit, that use the MSE when updating weights, every local minimum is also a global minimum. However, when using hidden layers, this might not always be the case.
These local minima might seem like a problem for artificial neural networks, but in practice they rarely are, according to theoretical and empirical results (Choromanska, Henaff, Mathieu, Arous, & LeCun, 2015). Instead, there are more saddle points. Here,

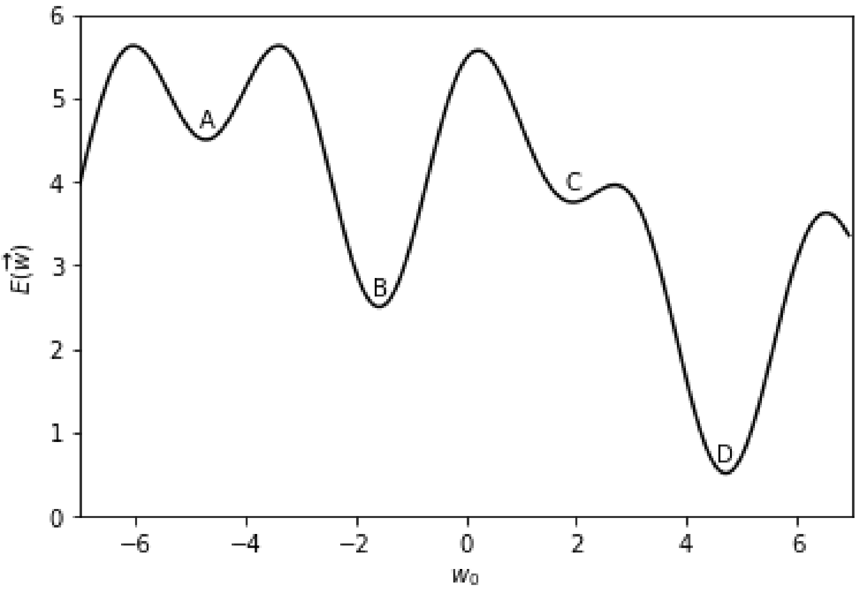

Figure 2.7: An example of an error surface using only one weight w_0. A, B and C are local minima, while D is the global minimum.

the gradient is zero and the error goes up when some weights are changed and goes down for others. Generally a lot of such points are present, but they all have the same value for the objective function (i.e. the error function).

A small value of η leads to slow convergence, while a high value can cause the algorithm to overstep a local minimum. Because of this, some algorithms gradually decrease the learning rate as the number of weight updates grows.

For a single weight vector component, the weight vector update becomes:

$$w_i \leftarrow w_i - \eta \frac{\delta E}{\delta w_i} \tag{2.3.4}$$

Of course, in order to update the weights, we first need to calculate the derivate of E w.r.t. each component w_i. As an example, we show the derivative when using a simple linear unit, which is defined as:

$$f(\overrightarrow{x}; \overrightarrow{w}) = \overrightarrow{x} \cdot \overrightarrow{w} \tag{2.3.5}$$

We then get the following derivation:

$$\begin{aligned}
\frac{\delta E}{\delta w_i} &= \frac{\delta}{\delta w_i}\frac{1}{N}\sum_{j=0}^{N}(y_j - o_j)^2 \\
&= \frac{1}{N}\sum_{j=0}^{N}\frac{\delta}{\delta w_i}(y_j - o_j)^2 \\
&= \frac{1}{N}\sum_{j=0}^{N}2(y_j - o_j)\frac{\delta}{\delta w_i}(y_j - o_j) \\
&= \frac{2}{N}\sum_{j=0}^{N}(y_j - o_j)\frac{\delta}{\delta w_i}(y_i - \overrightarrow{x_i} \cdot \overrightarrow{w}) \\
&= \frac{2}{N}\sum_{j=0}^{N}(y_j - o_j)(-x_{ij})
\end{aligned} \tag{2.3.6}$$

Where x_{ij} is the i'th input component of training example j and $o_j \equiv f(x_j; \overrightarrow{w})$. Instead of just using a linear unit, the function f can of course be more complex and use for example one of the activation functions from Section 2.2. However, we can now see that we cannot use the perceptron unit because it is not differentiable over its whole domain.

Using the resulting formula of Equation 2.3.6 along with Equation 2.3.4, we can also define how to update each component of the weight vector:

$$w_i \leftarrow w_i + \eta\frac{2}{N}\sum_{j=0}^{N}(y_j - o_j)x_{ij} \tag{2.3.7}$$

As we can see, for each weight vector, we need to apply the model to the input of each training example. For this reason, this version of gradient descent is also often called *batch gradient descent*. Of course, the outputs only need to be computed once per weight vector update as they do not depend on which specific weight vector component that we are updating.

Still, each time using every training example for the weight vector update can lead to slow convergence to a local minimum. Furthermore, in case of multiple minima in the error surface, it is possible that the gradient descent algorithm does not stop at a global minimum (Mitchell, 1997).

2.3.2 Stochastic gradient descent

A popular variation of batch gradient descent that tries to solve the previously mentioned issues is *stochastic gradient descent*. Instead of summing using all training examples, we apply a weight update using only one training example. To do this, we use a different error function for each example j:

$$E_j(\overrightarrow{w}) = (y_j - o_j)^2 \tag{2.3.8}$$

The update for a single weight vector component when using a linear unit is then:

$$w_i \leftarrow w_i + 2\eta(y_j - o_j)x_{ij} \tag{2.3.9}$$

The idea here is that these weight updates, when having iterated over all the training examples, will be a decent approximation relative to using our original error function. Note that the update using one training example affects the error of the next training example.

By making the learning rate small enough, usually smaller than with batch gradient descent, it is possible to approximate the result of batch gradient descent arbitrarily closely (Mitchell, 1997). It is also computationally cheaper because each time we only handle one training example. Additionally, stochastic gradient descent can sometimes avoid being stuck in local minima because it uses various $\nabla E_j(\overrightarrow{w})$ instead of just $\nabla E(\overrightarrow{w})$ to move in the hypothesis space.

2.3.3 Backpropagation

The backpropagation algorithm uses gradient descent to learn weights of a multilayer network with possibly multiple units in each layer (Rumelhart, Hinton, & Williams, 1986).

We say that, for multilayer networks, $L > 2$, with L the number of layers in the network. This means that there are other layers besides the input and output layer. Such a network was already depicted in Figure 2.1.

As multiple output units are also possible, we need a new training error measure that sums over all of the output units of the network:

$$E(\overrightarrow{w}) \equiv \frac{1}{N}\sum_{i=0}^{N}\sum_{k \in outputs}(y_{ki} - o_{ki})^2 \tag{2.3.10}$$

Where *outputs* are the output units of the network and y_{ki} and o_{ki} is the value of the k'th output unit of respectively the i'th training example and network output.

The difference in calculating the gradients in the weights is that it depends on previous layers that aren't input layers. That is, starting from the second non-input layer, the input to a unit comes, besides the weights, from other units of which the value itself is also calculated using weights. Thus, units in some layers can depend on units in multiple previous layers. This must also be taken into account when calculating the gradient for the output of those units. To do this, we can use the chain rule for the gradient of a single weight component:

$$\begin{aligned}\frac{\partial E_d}{\partial w_{ji}} &= \frac{\partial E_d}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}} \\ &= \frac{\partial E_d}{\partial net_j} x_{ji}\end{aligned} \tag{2.3.11}$$

Where E_d is the error for a training example d, x_{ji} is the i'th input to unit j, w_{ji} is the weight associated with this input and $net_j = \sum_i w_{ji}x_{ji}$. The further derivation for all the weights of the network is described in Mitchell, 1997, Chapter 4. For an artificial neural network with 1 hidden layer, we get the pseudo-code shown in Algorithm 1.

Algorithm 1: Backpropagation algorithm. For simplicity, the squared error ($E_j(\overrightarrow{w}) = \frac{1}{2}(y_j - o_j)^2$ for training example j) was used. Source: Mitchell, 1997, Chapter 4.

```
   Input: training_examples, neural_network, $\eta$
1  // Assume a network with $n_{in}$ input units $n_{hidden}$ hidden units and $n_{out}$ output
     units.
2  // Assume (randomly) instantiated weights.
3  repeat
4      for ($\overrightarrow{x}$, $\overrightarrow{y}$) in training_examples do
5          Propagate $\overrightarrow{x}$ through the network and receive values for all the output
             units: $o_u$ with $u$ an output unit of the network
6          for each network output unit $k$ do
7              Calculate its error term:
8              $\delta_k \leftarrow o_k(1 - o_k)(y_k - o_k)$
9          end
10         for each hidden unit $h$ do
11             Calculate its error term:
12             $\delta_h \leftarrow o_h(1 - o_h)\sum_{k \in outputs} w_{kh}\delta_k$
13         end
14         for each network weight $w_{ji}$ do
15             Do an update:
16             $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$
17             where
18             $\Delta w_{ji} = \eta\delta_j x_{ji}$
19         end
20     end
21 until a termination condition is met
```

2.3.4 Extensions and improvements

2.3.4.1 Different loss functions

Besides just the difference between the output of the network and the expected output, additional penalties can be added to indirectly influence the values of the weights. This is generally done in order to prevent overfitting. Overfitting means that the performance of the algorithm on the data on which we trained (called *training data*) is significantly better than the performance on held-out data on which we didn't train (called *test data*). The algorithm captures too much details or noise in the training data. Using penalties, we can force the algorithm to focus on generalization rather than specialization.
Here, we typically use the term *loss function* instead of an error function. This loss function can contain the error as we already described, along with other functions (typically also called loss functions).

A popular loss function is called the L_1 norm (also called Lasso) (Tibshirani, 1996). For each weight w, we add the value $\lambda|w|$ to the loss. The L_1 norm is then defined as such:

$$L_1(\overrightarrow{w}) = \lambda \sum_{i=1}^{k} |w_i| \tag{2.3.12}$$

Where k is the number of weights of the network and λ is a parameter determining the regularization strength. As the loss function, we then get:

$$L(\overrightarrow{w}) = E(\overrightarrow{w}) + L_1(\overrightarrow{w}) \tag{2.3.13}$$

This norm has the property of leading to sparse weight vectors (Bach, Jenatton, Mairal, & Obozinski, 2012). This means that a lot of weights are zero or close to zero and the focus is on finding a subset of the must important units.

Instead of using the absolute value, the L_2 norm sums over squared weights. It has the following form:

$$L_2(\overrightarrow{w}) = \frac{1}{2} \sum_{i=1}^{k} w_i^2 \tag{2.3.14}$$

The fraction $\frac{1}{2}$ is often used such that the derivative is λw instead of $2\lambda w$. The L_2 loss focuses more on penalizing high weights. It leads to diffusion of the weights and can indirectly force the network to use all the units of the network to form the correct output.

The L_1 norm and L_2 norm can also be combined, called *elastic net regularization* (Zou & Hastie, 2003):

$$L_E(\overrightarrow{w}) = L_1(\overrightarrow{w}) + L_2(\overrightarrow{w}) \tag{2.3.15}$$

$$= \sum_{i=1}^{k} \lambda_1 |w_i| + \lambda_2 w_i^2 \tag{2.3.16}$$

2.3.4.2 Different weight updates

Here we discuss methods that, given an error function, update the weights in different ways.

One of the most common extensions to backpropagation is to add a fraction of the previous weight update to the new weight update. This extension is called *momentum.* To apply this, we change the equation on line 18 of Algorithm 1 to the following:

$$\Delta w_{ji}(t) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(t-1) \tag{2.3.17}$$

Where $\Delta w_{ji}(t)$ is a weight update executed at iteration t and α, with $0 \leq \alpha < 1$, which determines how much of the previous weight update we want to carry to the new one. When $\alpha = 0$, we get again the regular weight update.

Thanks to momentum, it is possible to get past regions where the gradient is zero and the weights would not change without momentum. This is analogous to a ball rolling down a surface but keeps rolling a bit on a flat surface afterwards. In the real world, an object in motion will stay in motion, unless there is a force applied to it. Here, that force is the gradient. It is also possible to get past small local minima and it can speed up learning when the gradient does not change in subsequent iterations. This is analogous to a ball that goes downhill and keeps gaining speed.

The weight update of Root Mean Square Propagation (RMSProp) is quite different from stochastic gradient descent and only uses the sign of the gradient to update the weights (Tieleman & Hinton, 2012). This allows us to escape from plateaus more quickly. Furthermore, the learning rate can be different for each weight. For example, we can multiply the learning rate of a certain weight by a factor greater than one if its gradient in the last 2 updates agreed on the sign, and multiply the learning rate by a value less than 1 otherwise. This means that, when the gradients agree on the direction, we give them more influence. If they are not consistent and disagree in subsequent steps, we decrease the influence. We can also put a limit on the learning rate as to not let it grow to excessively low or high amounts.

However, by just using an adaptive learning rate, we can have the problem of weights themselves growing too high in unwanted situations. For example, when a gradient is for 9 subsequent steps a small positive amount (e.g. 0.1) and once a large negative amount (e.g. -0.9), we don't want to change the weights a lot because they balance each other out. Using the current algorithm however, the learning rate and as a result the weights would become too high because we only look at the signs and not the magnitude of the gradients.
Right now, we are dividing the gradient by its absolute value in order to just get $+1$ or -1 depending on the sign. It is however better to divide each weight by a moving

average of the squared weight using the gradients of the past steps:

$$MeanSquare(w,t) = \gamma \cdot MeanSquare(w,t-1) + (1-\gamma) \cdot \left(\frac{\partial E}{\partial w(t)}\right)^2 \tag{2.3.18a}$$

$$w(t+1) = w(t) - \frac{\alpha}{\sqrt{MeanSquare(w,t)}} \frac{\partial E}{\partial w(t)} \tag{2.3.18b}$$

Where $w(t)$ is a weight at time step t, $\frac{\partial E}{\partial w(t)}$ is its gradient, α is the learning rate and w_{t+1} is the new weight value. The past squared gradients are decayed by a value γ. Usually $\gamma = 0.9$. It is also possible to add a small number ϵ (e.g. $\epsilon = 10^{-8}$) to the denominator in Equation 2.3.18b to avoid division by zero.

Adaptive Moment Estimation (Adam) (Kingma & Ba, 2014) also uses an adaptive learning rate for each weight and an exponentially decaying average of previous squared gradients (also called the second moment). In addition, an exponentially decaying average of past gradients (also called the first moment) is also stored:

$$Mean(w,t) = \beta_1 \cdot Mean(w,t-1) + (1-\beta_1) \cdot \frac{\partial E}{\partial w(t)} \tag{2.3.19a}$$

$$MeanSquare(w,t) = \gamma \cdot MeanSquare(w,t-1) + (1-\gamma) \cdot \left(\frac{\partial E}{\partial w(t)}\right)^2 \tag{2.3.19b}$$

The authors of this method observed that the two previous equations are biased towards zero, which results in high weight updates. To alleviate this problem, the use bias-corrected values:

$$\widehat{Mean(w,t)} = \frac{Mean(w,t)}{1-\beta_1^t} \tag{2.3.20a}$$

$$\widehat{MeanSquare}(w,t) = \frac{MeanSquare(w,t)}{1-\beta_2^t} \tag{2.3.20b}$$

The weight update then becomes:

$$w(t+1) = w(t) - \frac{\alpha}{\sqrt{MeanSquare(w,t)}+\epsilon} Mean(w,t) \tag{2.3.21}$$

2.3.4.3 Dropout

Dropout is a way to prevent overfitting and is mostly applied to deep learning methods, which are explained in Chapter 4. It does so by randomly leaving out some visible or hidden units. As a result, all incoming and outgoing connections to these temporarily removed units are also removed and the algorithm cannot rely on these connections. The simplest way of determining if a unit has to be removed is by removing it with a certain probability p, e.g. $p = 0.5$. This value can also be determined by validation. As each unit can be present or not when using the network, for n units there are 2^n

possible networks. By learning each time using a possibly different network, we have a way of combining multiple models where each model is trained only a few times and most weights are shared between other models. Dropout is only used when training. In the testing phase, no units are removed. Here, however, we multiply the outgoing weights of each unit by the probability p. This is done to make sure that the *expected* output for any hidden unit is the same as the actual output at test time.

Chapter 3

Reinforcement learning

In reinforcement learning, one tries to find which action to take in a certain state of the environment in order to maximize a possibly delayed numerical reward.
For chapters 3.1 to 3.5 and 3.7, we will closely follow the explanation by Sutton and Barto, 1998.

3.1 Basics

Reinforcement learning problems can be formulated in the form of a Markov Decision Process (MDP). This is four-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$. $\mathcal{S}$ is the state space and contains all the possible states of the environment. The action space, $\mathcal{A}$, denotes all the possible actions that the agent can take in the environment. $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the transition function and defines the probabilities for being in next state given a state and an action. $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathcal{R}$ gives probabilities for rewards when taking an action in a certain state. A transition is then a tuple of a state, an action taken at that state, the received reward by taking that action (determined using $\mathcal{R}$) and the next state (determined using $\mathcal{P}$): (s_t, a_t, r_t, s_{t+1}). A sequence of transitions is called a trajectory.

A reinforcement learning algorithm does not know on beforehand which actions are optimal. As such, this must be discovered by trial-and-error. Subsequently, it can change its policy to increase the achieved rewards. This policy, denoted by π, defines for each state and action a probability $\pi(s, a)$, which denotes the probability of taking action a when in state s: $\pi_t(s, a) = P(a_t = a | s_t = s)$. Note that exactly one action has to be taken in a state and that the action probabilities for one state sum to 1: $\sum_{a \in \mathcal{A}} \pi_t(s, a) = 1$. Because feedback in the form of numerical rewards may not be immediate, a reinforcement learning algorithm must be able to backtrack which actions in certain situation lead to the reward that the reinforcement learning algorithm received.

Reinforcement learning algorithms often use a state-value function $V(s)$. This function says how good it is to be in a certain state s by stating the expected return in the future starting from state s, where the return is the discounted rewards. This expected value when starting in state s and following a specific policy π is denoted as $V^\pi(s)$:

$$V^\pi(s) = \mathop{\mathbb{E}}_\pi\{R_t|s_t = s\} = \mathop{\mathbb{E}}_\pi\Big\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s\Big\} \tag{3.1.1}$$

Where γ is a discount applied over time to past rewards. The optimal policy π is the one that leads to the highest $V^\pi(s)$ for all $s \in \mathcal{S}$. The resulting state-value function is $V^*(s) = \max_\pi V^\pi(s)$ for all $s \in \mathcal{S}$.

It is also possible to define a value for taking an action in a certain state, which is the action-value function $Q(s, a)$. $Q^\pi(s, a)$ is the expected return after taking action a in state s:

$$Q^\pi(s,a) = \mathop{\mathbb{E}}_\pi\{R_t|s_t = s, a_t = a\} = \mathop{\mathbb{E}}_\pi\Big\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s, a_t = a\Big\} \tag{3.1.2}$$

Similarly as with $V^*(s)$, $Q^*(s, a)$ is the action-value function we have when applying the optimal policy.
Actions can be selected by for example ϵ-greedy or softmax. In ϵ-greedy action selection, the action with the highest $Q(s, a)$ is selected with probability $1-\epsilon$ and a random action otherwise.
Softmax chooses an action a when in state s with the following probability:

$$p(s,a) = \frac{e^{Q(s,a)/\tau}}{\sum_{b=1}^{n} e^{Q(s,b)/\tau}} \tag{3.1.3}$$

Where τ is called the temperature and controls the balance between exploration and exploitation. For low values, the best actions are highly probable. For $\tau \to 1$, the probability distribution becomes uniform.

3.2 Dynamic programming

In Dynamic programming, the whole model of the problem is known. A such, we have the complete probability distribution of all the possible transitions. For example, to evaluate a policy π:

$$\begin{aligned} V^\pi(s) &= \mathop{\mathbb{E}}_\pi\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \mid s_t = s\} \\ &= \mathop{\mathbb{E}}_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s\} \\ &= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')] \end{aligned} \tag{3.2.1}$$

Here, $\pi(s, a)$ is the probability of taking action a in state s. This $V^\pi(s)$ can be computed using iterative updates for every s:

$$\begin{aligned} V_{k+1}(s) &= \mathop{\mathbb{E}}_\pi\{r_{t+1} + \gamma V_k(s_{t+1})|s_t = s\} \\ &= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V_k(s')] \end{aligned} \tag{3.2.2}$$

This iteration stops when this state-value function has converged. As can be seen, every possible next states is used in the computation instead of just a sample. Because of this, this kind of updates is called a full backup. Note that we use an estimate of the value function to update the estimate itself. This is called bootstrapping.
The optimal state-value function $V^*(s)$ and state-action-value function $Q^*(s,a)$ can be calculated using the following formulas:

$$\begin{aligned} V^*(s) &= \max_a E\{r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\} \\ &= \max_a \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V^*(s')] \end{aligned} \tag{3.2.3}$$

$$\begin{aligned} Q^*(s,a) &= E\{r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') | s_t = s, a_t = a\} \\ &= \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^*(s', a')] \end{aligned} \tag{3.2.4}$$

3.3 Monte Carlo and Temporal-Difference

Unlike dynamic programming, model-free methods don't require complete knowledge of the environment. Instead, only sample transitions are needed. This way, 2 problems with dynamic programming can be solved. First, the model may be large, which makes it infeasible to use for computing for example an optimal policy. Second, in real world problems, a complete model of the problem may not be available. It may for example be unknown what is the probability of ending in a certain state when taking a certain action from a start state.
Monte Carlo methods collect sample returns and average them in order to approximate a value function. The incremental implementation for approximating V is as follows:

$$V(s_t) \leftarrow V(s_t) + \alpha\big[R_t - V(s_t)\big] \tag{3.3.1}$$

Where R_t is the actual return of an action and α is a constant step-size parameter. As was shown in equation 3.1.1, to compute R_t we need all the future rewards until the end of the episode.
Temporal-Difference (TD) learning tries to solve this by bootstrapping techniques like in dynamic programming. To do this, in the update we replace the full return using the observed reward and the estimate of the value of the next state. This is known as TD(0):

$$V(s_t) \leftarrow V(s_t) + \alpha\big[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\big] \tag{3.3.2}$$

These estimates can then be used for acting in an environment.
Sarsa uses TD estimates for on-policy control. Because it is on-policy, we must estimate the action-value function $Q^\pi(s,a)$ for the current behavior policy π and for all states s and actions a. We consider transitions from state-action pair to state-action pair instead of transitions from state to state. The update is as follows:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\big[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\big] \tag{3.3.3}$$

We can then choose the action with the highest $Q^\pi(s_t, a_t)$, using ϵ-greedy etc. Another control method called Q-learning also uses TD estimates but doesn't use the policy that is followed to estimate Q^*. This is called an off-policy algorithm. Here, we do a backup using the action that has the highest Q value at the next state:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\big] \tag{3.3.4}$$

Note that the policy still can influence which state-action pairs are visited and updated. To guarantee finding the optimal behavior, it is required that all pairs continue to be updated.

3.4 Eligibility traces

Eligibility traces are used in order to only influence eligible states or actions when a certain action is taken.
Monte Carlo methods perform a backup for each state based on the entire sequence of observed rewards from that state until the end of the episode. The backup of more simple methods is only based on the next reward of a state, using the state value one step later as a proxy for the remaining rewards. An intermediate method would perform a backup based on an intermediate number of rewards. Then, we use the rewards of the intermediate steps and the estimated value of the last state. A visualization can be seen in Figure 3.1. The n-step target can be formulated as:

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n}) \tag{3.4.1}$$

There, we treat the terminal state as a state that always transitions to itself with zero reward. This way of treating an episodic task the same as a continuing task doesn't influence the result, even if we discount the returns. As such, all n-step returns that last up to or past termination have the same value as the complete return.
The increment to $V_t(s_t)$ is then defined by:

$$\Delta V_t(s_t) = \alpha\big[R_t^{(n)} - V_t(s_t)\big] \tag{3.4.2}$$

In on-line updating, the updates are done during the episode, as soon as a $\Delta V_t(s_t)$ is computed. In off-line updating, these increments are set aside and applied after the episode is done.
n-step TD methods are rarely used because they are inconvenient to implement. To compute n-step returns, you have to wait n steps to observe the resultant rewards and states.

We can also take the weighted average of n-step return. This is called the forward view or theoretical view. The requirement is that the weights sum to 1. TD(λ) is a particular way of doing this. The λ-return is defined by:

$$R_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)} \tag{3.4.3}$$

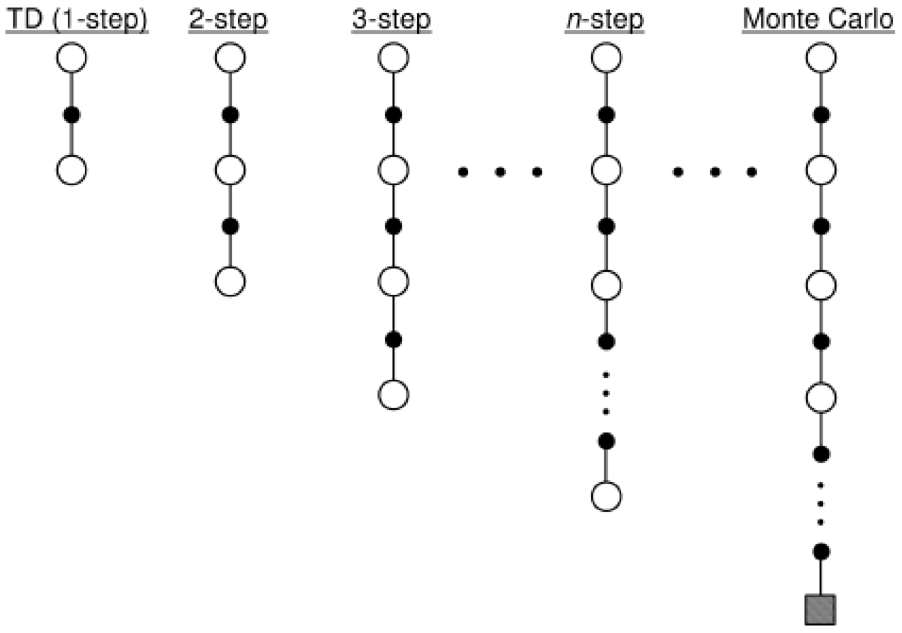

Figure 3.1: Returns each based on a different amount of rewards and by using the state value as a proxy afterwards. This is not needed for Monte Carlo methods as they use all the future rewards. Source: Sutton and Barto, 1998.

After a terminal state has been reached, all subsequent n-step returns are equal to R_t, so the $(1-\lambda)$ isn't applied. When $\lambda = 1$, we will only have the conventional return R_t. The increment, $\Delta V_t(s_t)$, is:

$$\Delta V_t(s_t) = \alpha\big[R_t^\lambda - V_t(s_t)\big] \tag{3.4.4}$$

In this forward view, we look for each state visited forward in time to all the future rewards and decide how best to combine them. After looking forward from and updating one state, we move to the next and don't work with that state anymore. Future states, however, are viewed and processed repeatedly. This way of thinking is visualized in Figure 3.2.

The backward view can be seen as a practical version of the forward view, as it achieves the same but is easier to implement. Here we have an additional memory variable associated with each state, the eligibility trace, denoted $e_t(s) \in \mathbb{R}^+$. On each step, they are decayed by $\gamma\lambda$, and the eligibility trace for the one state visited on the step is incremented by 1. The increment $\Delta V_t(s_t)$ is then defined as such:

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t) \tag{3.4.5a}$$

$$\Delta V_t(s) = \alpha \delta_t e_t(s), \qquad \text{for all} \quad s \in S \tag{3.4.5b}$$

Here we work backwards and update backward to each prior state according to the state's eligibility trace at that time. This is visualized in Figure 3.3. If we set $\lambda = 0$,

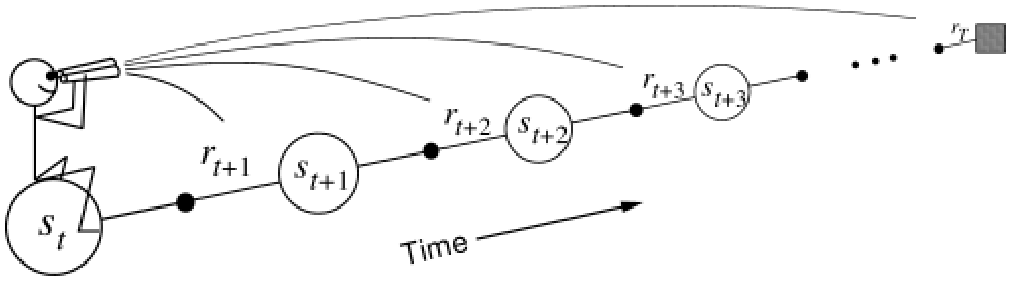

Figure 3.2: The forward view, where each state value is updated by looking forward to the states and their rewards that follow. Source: Sutton and Barto, 1998.

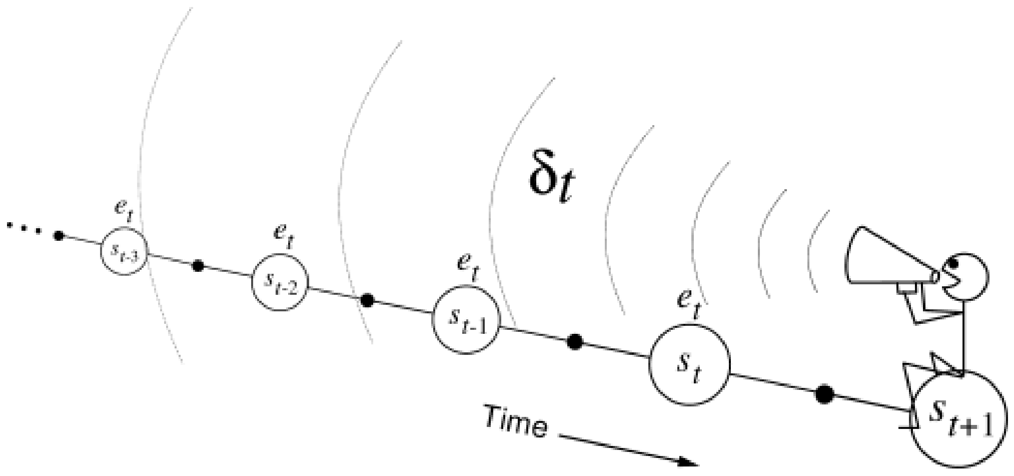

Figure 3.3: The backward view, where the value of visited states are updated based on their eligibility values. Source: Sutton and Barto, 1998.

then we only update the trace for s_t, thus getting TD(0). For larger values, with $\lambda < 1$, more of the preceding states are changed. The more distant states are changed by a smaller amount because the eligibility trace is smaller. This way, they are blamed less for the TD error.
If $\lambda = 1$ the credit given to earlier states falls only by γ per step. If $\lambda = 1$ and $\gamma = 1$, then there is no decay at all and we achieve the Monte Carlo method for an undiscounted episodic task. This is known as TD(1). This TD(1) is more general however because it cannot only be applied to episodic tasks, but also to discounted continuing tasks. It can also be performed incrementally and on-line, while Monte Carlo methods have to wait until the episode is over. If the Monte Carlo control method does something bad, its behavior cannot change during the same episode, while on-line TD(1) can (using n-step). Both methods achieve the same weight update.

To combine TD(λ) and Sarsa, called Sarsa(λ), we need eligibility traces for each state-action pair: $e_t(s,a)$. As such, we do updates like this:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\delta_t e_t(s,a), \qquad \text{for all} \quad s,a \tag{3.4.6}$$

where

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t) \tag{3.4.7}$$

and

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a) + 1 & \text{if } s = s_t \text{ and } a = a_t; \\ \gamma\lambda e_{t-1}(s,a) & \text{otherwise.} \end{cases} \qquad \text{for all } s,a \tag{3.4.8}$$

The resulting algorithm can be seen in Algorithm 2:
Two methods exist to combine TD(λ) and Q-learning and thus getting Q(λ): There

Algorithm 2: Sarsa(λ). Source: Sutton and Barto, 1998

```
Initialize $Q(s,a)$ arbitrarily
$e(s,a) \leftarrow 0$ for all $s,a$
for each episode do
    Initialize s,a
    repeat
        Take action $a$, observe reward $r$ and new state $s'$
        Choose action $a'$ from $s'$ with action selection policy using $Q$
        $\delta = r + \gamma Q(s',a') - Q_t(s,a)$
        $e(s,a) \leftarrow e(s,a) + 1$
        for all $s,a$ do
            $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
            $e(s,a) \leftarrow \gamma\lambda e(s,a)$
        end
        $s \leftarrow s'$; $a \leftarrow a'$
    until $s$ is terminal
end
```

exists 2 different methods: Watkins' Q(λ) and Peng's Q(λ).
Here, we must cut off the look ahead until the first exploratory action instead of the episode's end. This is done because this exploratory action doesn't have any relationship with the greedy policy. Recall that Q-learning is an off-policy method and the policy learned about is not necessarily the same as the one used to select actions. As such, it can learn about the greedy policy while following a policy involving exploratory (suboptimal) actions.

For Watkins' Q(λ), if a_{t+n} is the first exploratory action, the longest backup is toward:

$$r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n \max_a Q_t(s_{t+n}, a) \tag{3.4.9}$$

Where we assume off-line updating.
In the backward view, we update the eligibility traces just like in Sarsa, with the only exception that we don't use the eligibility trace of the previous time step when a sub-optimal (exploratory) action is taken. We get the following result:

$$e_t(s,a) = I_{ss_t} * I_{aa_t} + \begin{cases} \gamma\lambda e_{t-1}(s,a) & \text{if } Q_{t-1}(s_t,a_t) = \max_a Q_{t-1}(s_t,a); \\ 0 & \text{otherwise} \end{cases} \tag{3.4.10}$$

Where I_{xy} is an identity indicator function, equal to 1 if $x = y$ and 0 otherwise. The Q update is defined by:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\delta_t e_t(s,a) \tag{3.4.11}$$

Where

$$\delta_t = r_{t+1} + \gamma \max_{a'} Q_t(s_{t+1},a') - Q_t(s_t,a_t) \tag{3.4.12}$$

The resulting pseudo-code can be seen in Algorithm 3.
Because exploratory actions happen often, backups won't be long and so eligibility traces

Algorithm 3: Watkins' Q(λ). Source: Sutton and Barto, 1998

```
Initialize $Q(s,a)$ arbitrarily
$e(s,a) \leftarrow 0$ for all $s,a$
for each episode do
    Initialize s,a
    repeat
        Take action $a$, observe reward $r$ and new state $s'$
        Choose action $a'$ from $s'$ with action selection policy using $Q$
        $a^* \leftarrow \arg\max_b Q(s',b)$
        if $a^* = a'$ then
            $a^* \leftarrow a'$
        end
        $\delta = r + \gamma Q(s',a^*) - Q_t(s,a)$
        $e(s,a) \leftarrow e(s,a) + 1$
        for all $s,a$ do
            $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
            if $a^* = a'$ then
                $e(s,a) \leftarrow \gamma\lambda e(s,a)$
            else
                $e(s,a) \leftarrow 0$
            end
        end
        $s \leftarrow s'$; $a \leftarrow a'$
    until s is terminal
end
```

won't have a lot of advantage anymore.
Peng's Q(λ) tries to solve this, being a hybrid of Sarsa(λ) and Watkin's Q(λ). Its component backups are neither off- nor on-policy. The earlier transitions are each on-policy and the last transition uses the greedy policy. As such, all but the last uses the actual experiences. Because of this, for a fixed non-greedy policy, Q_t converges to neither Q^π nor Q^*, but some hybrid between the 2. If the policy is more greedy, the method may still converge to Q^*.

Replacing traces are a modified kind of eligibility traces that can yield a slightly better performance. With the traditional kind of traces (accumulating traces), the trace of a state is augmented by 1 when visiting it. With replacing traces, however, they are set to 1. Thus, we get the following update:

$$e_t(s) = \begin{cases} 1 & \text{if } s = s_t \\ \gamma\lambda e_{t-1}(s) & \text{otherwise} \end{cases} \tag{3.4.13}$$

Prediction or control algorithms using this are called replace-trace methods.

$$e_t(s,a) = \begin{cases} 1 + \gamma\lambda e_{t-1}(s,a) & \text{if } s = s_t \text{ and } a = a_t; \\ 0 & \text{if } s = s_t \text{ and } a \neq a_t; \\ \gamma\lambda e_{t-1}(s,a) & \text{if } s \neq s_t. \end{cases} \quad \text{for all } s, a \tag{3.4.14}$$

Another possible improvement if set correctly is a variable λ, which can be different at each time step t. For example, it may depend on the current state: $\lambda_t = \lambda(s_t)$. When we are sure about the estimate of the state s_t, it can be set to zero so we don't have to use estimates of following states anymore. By setting it to 1, we can achieve the opposite.

3.5 Bootstrapping

Bootstrapping is a technique where we update a value estimate based on other value estimates. This is done by TD and dynamic programming methods, but not by Monte Carlo methods. TD(λ) is a bootstrapping method when $\lambda < 1$, but not when $\lambda = 1$. Although the latter involves bootstrapping within an episode, afterwards the effect over a complete episode is the same as the non-bootstrapping Monte Carlo update. Bootstrapping methods such as TD(λ) are harder to combine with function approximation because they only find near-minimal MSE solutions (only for the on-policy distribution) instead of the minimal MSE using for example linear, gradient-descent function approximation for any distribution of training examples, P.
The restriction of convergence is especially a problem for off-policy methods such as Q-learning and dynamic programming methods because they do not backup states (or state-action pairs) with exactly the same distribution as the distribution of states we encounter when following the policy of which we want to estimate the value function.

Off-policy bootstrapping combined with function approximation can even lead to divergence and infinite MSE.

Bootstrapping looks like a bad idea because non-bootstrapping methods are more easy and reliably used with function approximation and can be applied over a broader range of conditions. Non-bootstrapping methods also achieve a lower error in approaching the value function, even when backups are done according to the on-policy distribution (Mitchell, 1997). In practice, however, bootstrapping methods seem to achieve better performance. When λ approaches 1, which is the non-bootstrapping case, the performance becomes much worse.

3.6 Policy gradient

Here, we parametrize the policy instead of the state value or state-action value function. Methods that parametrize the policy tend to have better convergence properties (Sutton, Mcallester, Singh, & Mansour, 1999). No value must be stored for every specific state and action. This is useful in problems where the state and/or action space is high-dimensional and/or continuous. Last, they can also learn policies where a probability is provided for every action. However, policy gradient methods also come with a few disadvantages. They typically converge to a local optimum instead of a global optimum. It is also rather hard to evaluate a policy and it can have a high variance.
The policy gradient has the form of $\pi_\theta(s,a)$. The way of measuring the quality of a policy (i.e. how well it performs in an environment) depends on the type of environment:

- Episodic environments: use the value of the start state:

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \underset{\pi_\theta}{\mathbb{E}}[v_1] \tag{3.6.1}$$

- Continuing environments: use the average value:

$$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s) \tag{3.6.2}$$

- or the average reward per time step:

$$J_{avR}(\theta) = \sum_s d^{\pi_\theta} \sum_a \pi_\theta(s,a) R_s^a \tag{3.6.3}$$

Where d^{π_θ} is the stationary distribution of the Markov chain for π_θ. The stationary distribution $\bar{\pi}$ of a Markov distribution P is a probability distribution such that $\bar{\pi} = \bar{\pi}P$. It can be thought of as denoting how much time is spent in each state. This way, values/rewards of states which are visited often are given a higher importance.
As such, we need to find the parameters θ that maximize $J(\theta)$. This optimization problem can be solved using approaches that don't use a gradient, such as hill climbing or

genetic algorithms. However, here we focus on gradient descent. In this case, we move the parameter values along the gradient using the quality function: $\Delta\theta = \alpha\nabla_\theta J(\theta)$, where α is the step-size parameter and $\nabla_\theta J(\theta)$ is the partial derivative for every dimension of θ. To estimate each partial derivative, we can compute the policy objective function $J(\theta)$ for a slightly changed value of θ, determined by ϵ. For each dimension k, this is:

$$\frac{\partial J(\theta)}{\partial\theta_k} \approx \frac{J(\theta + \epsilon u_k) - J(\theta)}{\epsilon} \tag{3.6.4}$$

where u_k is a one-hot vector with a value of 1 in dimension k and 0 elsewhere. This kind of gradient is called a Finite Difference gradient. It is a simple, black-box way that works for every policy (because we don't need the derivative), but it is inaccurate and inefficient. We also need to determine the value of ϵ ourselves, which influences the accuracy.

Instead, we will solve the problem analytically. We assume that the policy is differentiable when it is non-zero and that we know the gradient.

Likelihood ratios use the following identity:

$$\begin{aligned}\nabla_\theta\pi_\theta(s,a) &= \pi_\theta(s,a)\frac{\nabla_\theta\pi_\theta(s,a)}{\pi_\theta(s,a)} \\ &= \pi_\theta(s,a)\nabla_\theta\log\pi_\theta(s,a)\end{aligned} \tag{3.6.5}$$

Writing it using the log is possible because $(\log f)' = \frac{f'}{f}$. The score function is $\nabla_\theta \log\pi_\theta(s,a)$. This can now be used with for example a softmax policy. Here, we weight actions using a linear combination of features: $\phi(s_a)^T\theta$. The probability of an action is then the following proportion:

$$\pi_\theta(s,a) \propto \mathrm{e}^{\phi(\mathrm{s,a})^{\mathrm{T}}\theta} \tag{3.6.6}$$

The score function is here defined as being:

$$\nabla_\theta\log\pi_\theta(s,a) = \phi(s,a) - \mathop{\mathbb{E}}_{\pi_\theta}[\phi(s,\cdot)] \tag{3.6.7}$$

When having a continuous action space, it is better to use a Gaussian policy, where the mean is a linear combination $\mu(s) = \phi(s)^T\theta$ and the variance can be either fixed or also parametrized. The policy is then $a \sim N(\mu(s), \sigma^2)$. The score function here is defined as being:

$$\nabla_\theta\log\pi_\theta(s,a) = \frac{(a-\mu(s))\phi(s)}{\sigma^2} \tag{3.6.8}$$

We can now use the likelihood ratios to compute the policy gradients for one-step Markov decision processes:

$$\begin{aligned} J(\theta) &= \mathop{\mathbb{E}}_{\pi_\theta}[r] \\ &= \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(s,a) R(s,a) \end{aligned} \tag{3.6.9a}$$

$$\begin{aligned} \nabla_\theta J(\theta) &= \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a) R(s,a) \\ &= \mathop{\mathbb{E}}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) r] \end{aligned} \tag{3.6.9b}$$

The Policy Gradient theorem states that we can use this for multi-step MDP's (Sutton et al., 1999). We can do this by replacing the reward with the state-action value:

$$\mathop{\mathbb{E}}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)] \tag{3.6.10}$$

Monte-Carlo policy gradients use this policy gradient theorem and update parameters using stochastic gradient descent. For this, a return v_t is used as an unbiased sample of $Q^{\pi_\theta}(s,a)$. The resulting algorithm is called *REINFORCE* (Williams, 1992) and is shown in Algorithm 4.

Algorithm 4: REINFORCE algorithm. Source: Sutton and Barto, 1998.

```
1 Initialize $\theta$ arbitrarily
2 for each episode $\{s_1, a_1, r_2, \ldots, s_{T_1}, a_{T_1}, r_T\} \sim \pi_\theta$ do
3     for $t = 1$ to $T - 1$ do
4         $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
5     end
6 end
7 return $\theta$
```

The problem with the Monte-Carlo policy gradient is that it has a high variance. This can be solved by using an actor-critic method. The critic is used to update the state-action value function parameters w, such that $Q_w(s,a) \approx Q^{\pi_\theta}(s,a)$, while the actor updates the policy parameters θ in the direction suggested by the critic. These actor-critic algorithms follow an approximate policy gradient:

$$\nabla_\theta J(\theta) \approx \mathop{\mathbb{E}}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q_w(s,a)] \tag{3.6.11}$$

$$\Delta\theta = \alpha \nabla_\theta \log \pi_\theta(s,a) Q_w(s,a) \tag{3.6.12}$$

The critic function can be seen of a way of evaluating how good the policy is for the current θ. A simple way of computing $Q_w(s,a)$ is simply a linear function of w and $\phi(s,a)$: $Q_w(s,a) = \phi(s,a)^T w$. The algorithm, named *QAC* is shown in Algorithm 5. For a function approximation to be compatible, it needs to fulfill 2 conditions:

Algorithm 5: QAC

```
1  Initialize $s, \theta$
2  Sample $a \sim \pi_\theta$
3  for each step do
4      Sample reward $r = R_s^a$; sample transition $s' \sim P_s^a$
5      Sample action $a' \sim \pi_\theta(s', a')$
6      $\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$
7      $\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$
8      $w \leftarrow w + \beta \delta \phi(s, a)$
9      $a \leftarrow a'$, $s \leftarrow s'$
10 end
```

- The value function approximator is compatible to the policy: $\nabla_w Q_w(s,a) = \nabla_\theta \log \pi_\theta(s,a)$
- The value function parameters w minimize the mean-squared error: $\epsilon = \mathbb{E}_{\pi_\theta}[(Q^{\pi_\theta}(s,a) - Q_w(s,a))^2]$

The policy gradient will then be exactly:

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q_w(s,a)] \tag{3.6.13}$$

To reduce the variance in the policy gradient we subtract the baseline function $B(s)$ from it. This doesn't change the expectation:

$$\begin{aligned} \mathop{\mathbb{E}}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) B(s)] &= \sum_{s\in S} d^{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(s,a) B(s) \\ &= \sum_{s\in S} d^{\pi_\theta} B(s) \nabla_\theta \sum_{a\in A} \pi_\theta(s,a) \\ &= 0 \end{aligned} \tag{3.6.14}$$

A good baseline function is the state value function: $B(s) = V^{\pi_\theta}$. When this is used as a baseline function, we can rewrite the policy gradient using the advantage function $A^{\pi_\theta}(s,a)$:

$$A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s) \tag{3.6.15}$$

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) A^{\pi_\theta}(s,a)] \tag{3.6.16}$$

This still reduces the variance. The critic should now estimate the advantage function. This can be done by estimating both $V^{\pi_\theta}(s)$ and $Q^{\pi_\theta}(s,a)$. With those 2 function approximators we then get:

$$V_v(s) \approx V^{\pi_\theta}(s) \tag{3.6.17}$$

$$Q_w(s,a) \approx Q^{\pi_\theta}(s,a) \tag{3.6.18}$$

$$A(s,a) = Q_w(s,a) - V_v(s) \tag{3.6.19}$$

For updating those values, we can again use temporal difference learning. The TD error is:

$$\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s) \tag{3.6.20}$$

This is an unbiased estimate of the advantage function:

$$\begin{aligned} \mathop{\mathbb{E}}_{\pi_\theta}[\delta^{\pi_\theta}|s,a] &= \mathop{\mathbb{E}}_{\pi_\theta}[r + \gamma V^{\pi_\theta}(s')|s,a] - V^{\pi_\theta}(s) \\ &= Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s) \\ &= A^{\pi_\theta}(s,a) \end{aligned} \tag{3.6.21}$$

As such, we can use this TD error to compute the policy gradient:

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\delta^{\pi_\theta}] \tag{3.6.22}$$

In practice, we use the function approximator of the critic function for the value function, using the critic parameters v:

$$\delta_v = r + \gamma V_v(s') - V_v(s) \tag{3.6.23}$$

3.7 Generalization and function approximation

Normally, if there are a lot of possible states, it will take a long time to learn the estimates of all states. It is even possible that, after some time, previously unseen states will be encountered. This problem is possible when using continuous variables, images, ... To solve this, we generalize states. As such, we can apply information of seen states to related states that haven't been visited yet. To do this, we combine standard reinforcement learning methods with generalization methods.

A well known kind of generalization methods is called function approximation: it takes examples of a desired function and tries to approximate it. This is supervised learning (the input is the original value and the output to predict is the generalized value). The used supervised learning algorithm needs to be able to handle non-stationary target functions, as learning must be able to occur on-line. The error for the algorithm to minimize is the mean-squared error (MSE) between the true value of the state and the approximated one:

$$MSE(\overrightarrow{\theta_t}) = \sum_{s \in S} P(s)\big[V^\pi(s) - V_t(s)\big]^2 \tag{3.7.1}$$

Where θ_t is a component of the model that the algorithm generated and P is a distribution weighting the errors of different states. As there are less components $\overrightarrow{\theta_t}$ than states, the flexibility for approximation is limited. Because of this, we use P to define the trade-offs of focusing on improving the approximation of some states at the expense of others. Usually, this distribution P is the same as the one from which the states in the

training examples are drawn from and thus the distribution of states at which backups are done. For minimizing the error over a certain distribution of states, it is of course preferred that the training examples come from the same distribution.
Another interesting distribution is the on-policy distribution, which describes the frequency with which states are encountered while the agent is interacting with the environment and selecting actions according to policy π. Minimizing the error over this distribution, we concentrate on states that actually occur while following the policy and ignoring others. Training examples for this distribution are also the easiest to get using Monte Carlo or TD methods because they generate backups from sample experience using the policy π.
For simple function approximators such as linear ones, the best MSE we find might also be the global optimum. This is however rarely possible for more complex ones, e.g. kernel based methods or artificial neural networks, and they may stop at a local optimum. For many cases of interest in reinforcement learning, convergence to an optimum, i.e. achieving the highest possible reward, does not occur.

One of the most widely used function approximation methods is based on gradient descent. Here, the parameter vector is a column vector with a fixed number of real valued components, also called a weight vector: $\overrightarrow{\theta_t} = (\theta_t(1), \theta_t(2), \ldots, \theta_t(n)^T$. $V_t(s)$ is a smooth differentiable function of $\overrightarrow{\theta_t}$ for all $s \in S$. At each time step t, we observe a new example $s_t \mapsto V^\pi(s_t)$. The order of the received states is not assumed to be the same as the order of gathering them from transitions in the environment. Even if we would give the exact $V^\pi(s_t)$, the function approximator has only limited resources and would not be able to approximate the function exactly. Thus, it must generalize.
Like already discussed, we assume that the states over which we want to minimize the MSE over come from the same distribution P as the from examples. We can then use gradient descent, explained in Section 2.3.1, to adjust the parameter vector in order to minimize the error:

$$\begin{aligned} \overrightarrow{\theta}_{t+1} &= \overrightarrow{\theta}_t - \frac{1}{2}\alpha\Delta_{\overrightarrow{\theta}_t}\big[V^\pi(s_t) - V_t(s_t)\big]^2 \\ &= \overrightarrow{\theta}_t + \alpha\big[V^\pi(s_t) - V_t(s_t)\big]\Delta_{\overrightarrow{\theta}_t} V_t(s_t) \end{aligned} \tag{3.7.2}$$

Where α is a positive step-size parameter and $\nabla_{\overrightarrow{\theta}_t}$ denotes the vector of partial derivatives for every function f.
If $V^\pi(s_t)$ is unavailable because we only have a noise-corrupted version or one with backed-up values, we can simple use v_t instead of $V^\pi(s_t)$:

$$\overrightarrow{\theta}_{t+1} = \overrightarrow{\theta}_t + \alpha\big[v_t - V_t(s_t)\big]\Delta_{\overrightarrow{\theta}_t} V_t(s_t) \tag{3.7.3}$$

If v_t is an unbiased estimate and so $E\{v_t\} = V^\pi(s_t)$ for each t, then $\overrightarrow{\theta}_t$ is guaranteed to converge to a local optimum under stochastic approximation conditions for decreasing the step-size parameter α. This is the case for Monte Carlo state-value prediction.

An important special case is when the approximate function V_t is a linear function of the parameter vector, $\overrightarrow{\theta}_t$. For every state s, there is a column vector of features $\overrightarrow{\phi}_s = (\phi_s(1), \phi_s(2), \ldots, \phi_s(n))^T$, with the same number of components as $\overrightarrow{\theta}_t$. The approximate state-value function is then given by:

$$V_t(s) = \overrightarrow{\theta}_t^T \overrightarrow{\phi}_s = \sum_{i=1}^{n} \theta_t(i)\phi_s(i) \tag{3.7.4}$$

The gradient with respect to $\overrightarrow{\theta}_t$ is then:

$$\nabla_{\overrightarrow{\theta}_t} V_t(s) = \overrightarrow{\phi}_s \tag{3.7.5}$$

As can be seen, this update is rather simple. Furthermore, there is only one optimum (or several ones which are equally good), $\overrightarrow{\theta}^*$. As a result, the method is guaranteed to converge to or near a local optimum.
Note that this linear form doesn't allow for the representation of interactions between features, for example when the presence of a certain features is good only if another feature is absent. For this, we need to introduce features that are combinations of feature values.

3.7.1 Coarse coding

Coarse coding is the representation of a state with features that overlap, for example a binary feature that is 1 when the coordinate given by a x and y feature lies in a circle. If 2 points A and B have circles "in common", there will be some generalization between them, as the features for both points for those circles will be 1. This is shown in Figure 3.4. The more features in common, the greater this effect. If the circles are small or large, generalization will be over respectively a short or large distance. The nature of the generalization is also affected by the shape of the features' receptive fields. This can be seen in Figure 3.5. The fineness of discrimination is however only determined by the total number of features.

In tile encoding, a form of coarse encoding, the fields of the features are grouped into exhaustive partitions of the input space, called tilings. These tilings are defined by ranges of values for state attributes that they cover. For one specific tiling, the state can only lie in the ranges of 1 tile. As such, maximally one feature is active in each tiling and the total number of features present is always the same as the number of tilings. This allows us to set the step-size parameter α (used in formula 3.4.5a) in an easy, intuitive way. We can for example choose $\alpha = \frac{1}{m}$, where m is the number of tilings. This results in exact one-trial learning. When the example $s_t \mapsto v_t$ is received, then the new value will be $V_{t+1}(s_t) = v_t$, whatever the prior value $V_t(s_t)$ was. For a slower change using for example $\alpha = \frac{1}{10m}$, one would move one-tenth of the way to the target in one update.
The weighted sum to make up the approximate value function is almost trivial to compute, as in tile coding only binary features are used. Instead of performing n multiplications and additions, we compute the indices of the $m << n$ present features and then

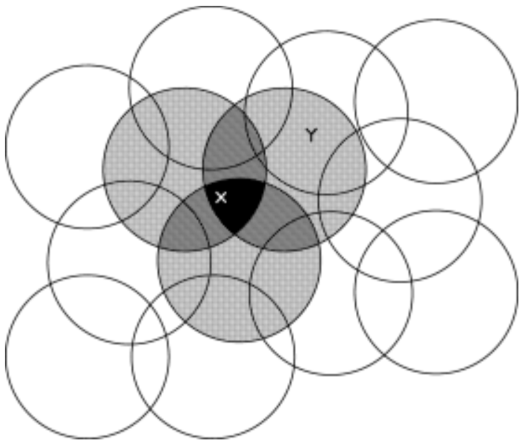

Figure 3.4: X and Y share 1 feature, as both points lie in the same receptive field, a circle. Thus, slight generalization from X to Y is possible. Source: Sutton and Barto, 1998.

add up the m corresponding components of the parameter vector. The eligibility trace computation is also easier because the components of the gradient $\nabla_{\vec{\theta}} V_t(s_t)$ are either 0 or 1.
An easy to compute form of tilings are grid-like ones. Different tilings may also be used, ideally offset by different amounts. The width and shapes should match the width of generalization that is expected to be optimal. The denser the tiling, the more fine-grained the desired function can be, but also the greater the computational cost of updating. The width or shape of each tile must not be uniform or regular-shaped (like a square) however. One can also use stripes, diamond shapes, . . . To reduce memory requirements, hashing can be used. A large tiling can be collapsed in such way into a much smaller set of tiles. This hashing produces tiles consisting of non-contiguous, disjoint regions randomly spread throughout the state space, but that still form an exhaustive tiling.
Radial basis functions are a generalization of coarse coding that allows for continuous-valued features: in the interval $[0, 1]$ instead of just 0 or 1. This way, we can reflect various degrees to which the feature is present. A typical RBF feature, i, has a Gaussian (bell-shaped) response $\phi_s(i)$ dependent only on the distance between the state and the feature's center state, c_i and relative to the feature's width σ_i:

$$\phi_s(i) = e^{-\frac{||s-c_i||^2}{2\sigma_i^2}} \tag{3.7.6}$$

The advantage of this method is that they produce approximate functions that vary smoothly and are differentiable. The disadvantage is that σ_i and c_i must be tuned manually and that non-linear RBF networks are computationally complex.

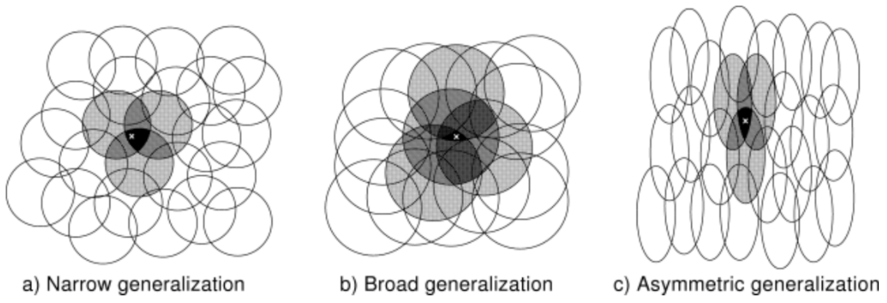

Figure 3.5: Generalization depends on the shape of the receptive fields of the features. When multiple features' receptive fields overlap, the generalization is broad and narrow when only few of them overlap. Source: Sutton and Barto, 1998.

Chapter 4

Deep learning

As was briefly mentioned in Section 3.7, artificial neural networks can be used as function approximators. However, feedforward artificial neural networks may lack capabilities to represent complex functions and might be unusable for certain kinds of input. They also lack granularity and can be influenced by variance in the input that is irrelevant for the task (e.g. in a classification task, shifted objects in a picture should still be classified correctly). Because of this, they are not suitable for complex tasks like image recognition, unless an adequate feature extractor is applied first. Generic methods like kernel methods can be applied but are not guaranteed to work well for the task.
Generally, feature extractors need to be developed manually in order to represent the aspects of the input that are important. This, however, requires more work and requires knowledge about the domain and the task.

In contrast, deep learning methods are able to learn multiple levels of representations of raw data without human guidance. Each level of features builds upon the previous one, starting from the input itself, and represents it in a more abstract way. These abstractions are generated as to recognize aspects of the input that are important for the task. Because of this, it is possible that small variations in the input have no influence on the abstraction and output. As these abstractions are learned automatically, there is no need anymore to create internal representations manually, which require more work as it is different for every kind of task. The generated abstractions may also recognize useful patterns in the data that might not be intuitive to a human and would not have been represented using the hand-crafted features.

In deep learning, the important aspects of the input can be detected automatically by combining different modules that have different functions in order to get higher abstractions and to eventually generate output. Most of these modules contain weights an can be trained, like explained in Section 2.3.
Here, we explain artificial neural networks capable of learning representations using multi-dimensional arrays and sequential data, following the explanation by LeCun, Bengio, and Hinton, 2015.

4.1 Convolutional neural networks

Convolutional neural networks are neural networks that are inspired by the way the visual cortex of an animal works. These networks get as input data in multi-dimensional arrays, such as 2-dimensional images with pixel intensities as the third dimension for example.
A convolutional neural network is structured as a series of stages, which are combinations of layers. In the first stages, convolutional layers and pooling layers are used. In a convolutional layer, units are organized in feature maps. Each of these units is connected to a patch in the feature maps of the previous layer through a matrix of weights that is called the *filter bank* or the *kernel*. The filter bank matrix then slides over the feature map, does an element-wise computation of the 2 matrices and sums the results. The filter bank is then moved by a certain amount, called the stride. All units in a specific feature map of a layer use the same filter bank. An example of this process is shown in Figure 4.1.

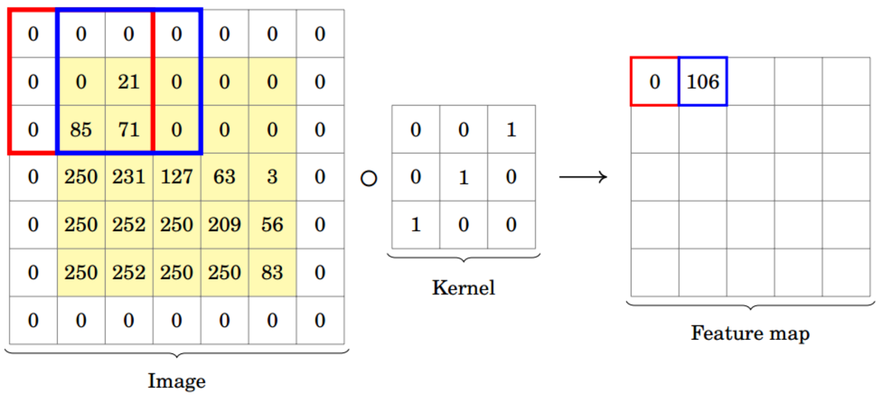

Figure 4.1: An example of a 3-by-3 filter applied to an image. The filter is first applied using an element-wise computation and summation on the red square shown in the image, resulting in the value on the right. Then, because the stride is 1, we move the filter one pixel to the right (shown in the blue square) and perform the same computation. Zeros are added in order to get a feature map of the same size as the input. Source: Pavlovsky, 2017.

These filters are used because local groups of values may be correlated and as such may have the same weights. This reduces the amount of weights that need to be trained. Furthermore, certain concepts and local statistics (of images) may be invariant to the location. A certain pattern or motif in an image might appear in different places of the image but may have the same meaning.
The result of this convolutional operation (which is linear) operation is again a feature map and can then be passed to a non-linear function such as a *ReLU*.

These convolutional layers are then combined with pooling layers. Instead of de-

tecting local conjunctions of features from the previous layer, pooling tries to merge semantically similar features into one. This is necessary because the positions of features that form a motif are not exact and can vary somewhat. It may not matter for example that an object is close or far in an image. To detect the motifs more reliably, we course-grain their features. The typical pooling units compute the maximum or average of a local patch of units in one or a few feature maps. The neighboring pooling units do the same for patches that are shifted by one or more rows and/or columns and as such creating an invariance to small shifts and distortions.
Popular convolutional networks, often using multiple layers of different types, include "AlexNet" (Krizhevsky, Sutskever, & Hinton, 2012), "LeNet" (LeCun, Bottou, Bengio, & Haffner, 1998) and "GoogLeNet" (Szegedy et al., 2014).
A typical layout of a convolutional neural network is shown in Figure 4.2, while a hierarchy of features extracted using a convolutional neural network is shown in Figure 4.3.

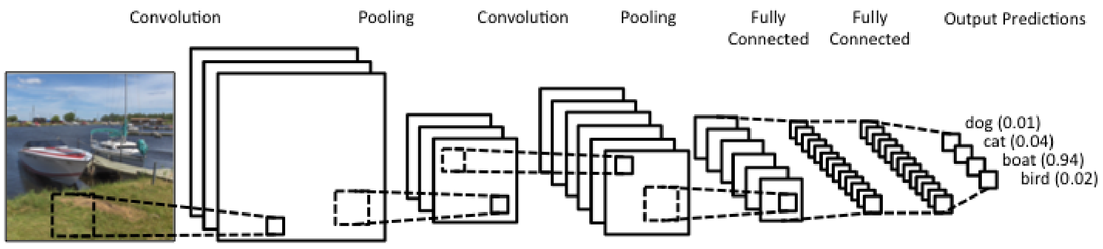

Figure 4.2: Layout of a typical convolutional neural network, which uses convolutional layers, pooling layers and fully connected layers in order to detect an object in an image. Source: https://www.clarifai.com/technology

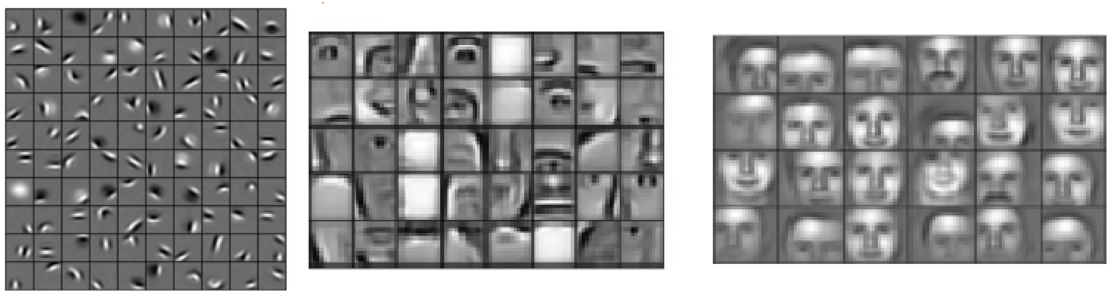

Figure 4.3: A hierarchy of features with as low-level features contours of faces. A more complex layer represents parts of faces. The final layer shows whole faces and can be used for classification or regression. Note that these features don't always have an intuitive meaning. Source: Lee, Grosse, Ranganath, and Ng, 2009.

4.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) are used to process sequential data such as speech and language. Here, an input sequence is processed one element at a time. The first RNNs used a state vector in their units that contains history about the past elements in the sequence. This is visualized in Figure 4.4.

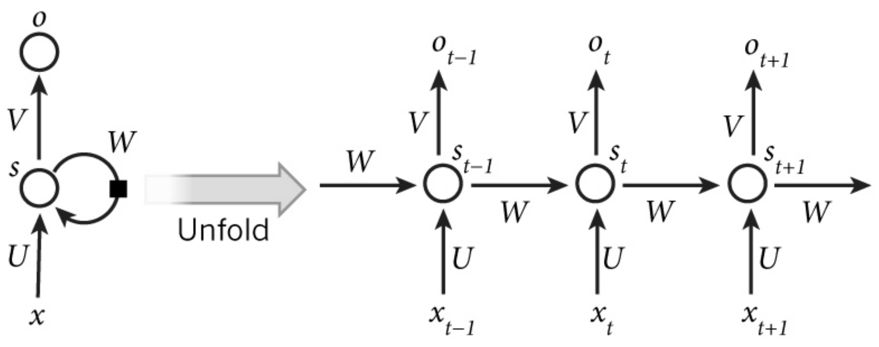

Figure 4.4: A recurrent neural network in which each state (hidden unit) is passed to the next one. x_t and o_t are respectively the input and output at time step t. s_t is the output of a hidden layer and depends on the input and the hidden unit at the previous time step: $s_t = U * x_t + W * s_{t-1}$. Note that the same weights U, V and W are used at each time step. Source: LeCun, Bengio, and Hinton, 2015.

Training them was a problem because of the vanishing or exploding gradients, as these gradients either shrink or grow at every time step. Theoretical and empirical evidence has shown that it is hard for these kind of networks to store information long enough and that they have difficulties to learn the long-term dependencies (Bengio, Simard, & Frasconi, 1994).

To learn RNNs, we also need to change the backpropagation algorithm in order to handle with the different time steps. Here, the error (for one training example) is just the sum of the errors at every time step: $e = \sum_t e_t$. The adapted backpropagation algorithm, called backpropagation through time (BPTT), also just sums up the gradient at every time step for every set of weights: $\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$. For the weights U and V of Figure 4.4, the gradients are computed just as in normal backpropagation. For W however, we depend on the previous state, which cannot simple be treated as a constant, and must be replaced as well. Because of this, we get:

$$\frac{\partial E_t}{\partial W} = \sum_{k=0}^{t} \frac{\partial E_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial s_t} \frac{\partial s_t}{\partial s_k} \frac{\partial s_k}{\partial W} \tag{4.2.1}$$

We see that training can be hard when the sequences are long, because we need to propagate back through the first time step.

For the *tanh* and *sigmoid* function, the derivative is 0 at both ends. This means that the gradients in other layers will also go towards zero and for small values in the matrix multiplications the gradient values are shrinking exponentially fast and become nearly zero after a few time steps. Because of this, steps far away won't contribute much to what you're currently computing, and as such the long-term dependencies are small. If the values of the partial derivatives are high however, we could get exploding gradients. These are however easier to detect and solve, because the value of the gradients will be too high to be presented by a variable in the programming language. The problem can also easily be solved by restricting the value in a certain range.
The problem of vanishing gradients can be solved by a better initialization of the weights, by regularization and by another non-linear function, namely the already explained *ReLU*. For this function, the derivative is either 0 (when $z < 0$) or 1 ($\frac{\partial z}{\partial z} = 1$).

A way of solving this is using a long short-term memory. This is a combination of different units and has an explicit memory. One of the units has a connection with itself and is used to accumulate or forget the stored information. It can be learned using another unit to know when to forget something and when to remember past information. First, the input is passed to an input gate i_t that determines how much of the input to let through into the memory. By also both adding and forgetting a part, determined using a forget gate f_t, the internal memory is updated. An output gate o_t then determines how much of the internal memory to output to the next layer. The architecture of an LSTM cell is shown in Figure 4.5.

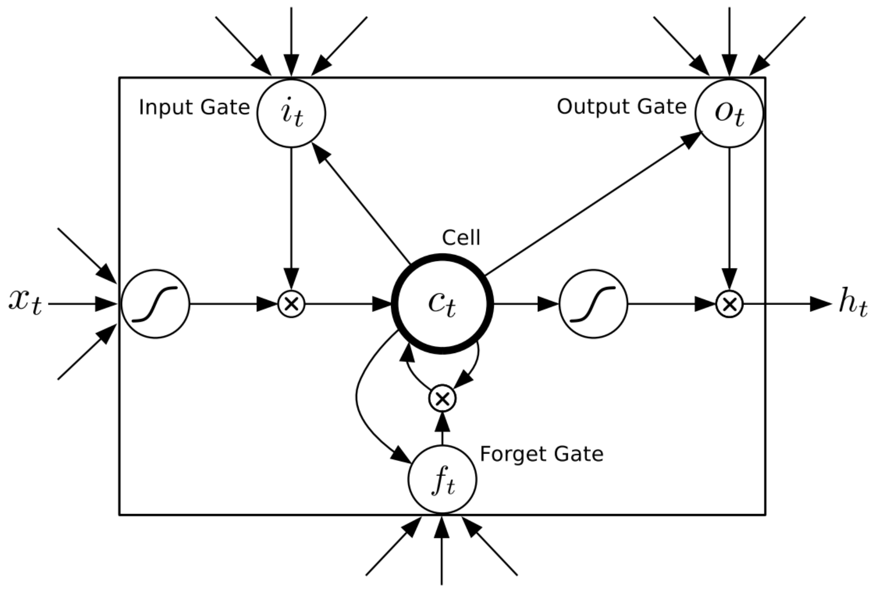

Figure 4.5: The architecture of an LSTM cell. Source: Graves, 2013.

Another kind of recurrent unit is the Gated Recurrent Unit (GRU), which was in-

troduced by Cho et al., 2014. It is simpler than LSTM but yields similar results. It also uses an internal memory, here called the *hidden state* and denoted by h. An update gate z determines how much of the previous memory we want to keep, depending on the previous hidden state and the input. A reset gate r says how much of the previous hidden state we want to keep in the new hidden state, depending on the hidden state and the input. The new hidden state is then a combination of the partly ”reset” previous hidden state and the input. The output is the sum of the previous hidden state and the new one, weighted by the reset gate.
By always setting the reset gate to 1 and the update gate to 0 we get again the classic RNN architecture.
The architecture of a GRU cell is visualized in Figure 4.6.

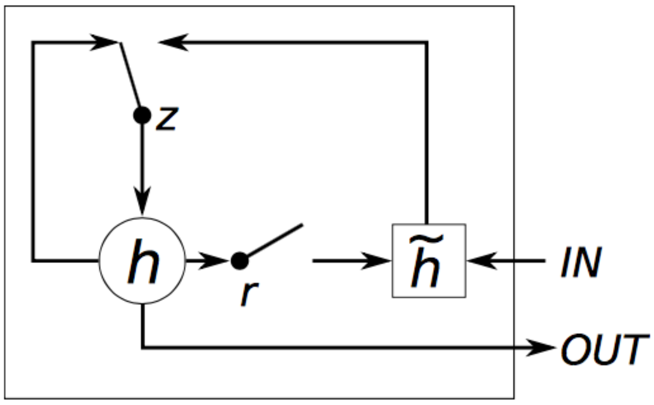

Figure 4.6: Gated Recurrent Unit cell. Source: Chung, Gülçehre, Cho, and Bengio, 2014.

Other networks with memory include a Neural Turing Machine, in which a tape-like memory is used from which the network can read or write.
Another kind is a memory network, in which a normal network is augmented by a associative memory.

Chapter 5

Deep reinforcement learning

In deep reinforcement learning, deep learning methods are used to generalize over states and to approximate functions that are part of reinforcement learning algorithms. Using this combination, we can learn to act in environments where the state is represented by a high-dimensional value, such as an image. Again, multiple levels of representations are built. Here, these representations are built in order to perform better in an environment.
Besides dealing with high-dimensional inputs, deep reinforcement learning algorithms must also be able to overcome instability issues. These are caused by subsequent observations being correlated. When learning while acting in the environment, learned knowledge can also be overwritten when the data distribution changes (called "catastrophic forgetting"), an issue that occurs for example when exploring a new part of the state space. Last, it is also possible that the function approximator parameters diverge.

First, we will discuss Deep Q-Network (DQN), which solves these problems by using a more stable target to learn on and learns on past experiences, leading to uncorrelated data. Afterwards, we will discuss Asynchronous Advantage Actor Critic (A3C). Instead of learning using past experiences, this algorithms obtains uncorrelated data and avoids catastrophic forgetting by learning in parallel with different actors on the same environment.

5.1 DQN

A Deep Q-Network (DQN), presented in Mnih et al., 2015, aims to combine deep learning with reinforcement learning. It applies a convolutional neural network to high-dimensional sensory inputs such as the pixels of a game screen in order to approximate an action-value function $Q^*(s, a)$. This is parametrized by the weights of the convolutional neural network, thus being $Q(s; a; \theta_i)$. Past experiences are also reused and the Q value function used to select action is only periodically updated to the most recently learned one in order to reduce instability.

This instability is caused by using a nonlinear function to approximate the action-value function and by the correlation between subsequent observations. Small changes in the Q value can lead to big changes in the policy and because of that the data distribution and the correlations between the action-values $Q(s,a)$ and the target values $r + \gamma \max_{a'} Q(s',a')$ can also change a lot. The action that is selected determines which states that will be observed next.

Experience replay solves the problem of having correlated subsequent observations and changing data distributions by randomizing used mini-batches of the data on which is learned. In each iteration, these random mini-batches of saved experiences are used for applying Q-learning updates. Experiences can be used multiple times, which results in an efficient use of the available data.
For the replay memory, all the encountered experiences can be used or only the N most recent ones. Here, however, it is possible that we forget experiences that may be important. Because we use uniform sampling to determine which experiences to use, we also give no extra importance (i.e. more probability) to more useful experiences.

A second improvement is to only update the used target function every C steps. This helps to reduce the correlation between the action-value function values and the target values, as for example $Q(s_t, a_t)$ can have an influence on $Q(s_{t+1}, a)$. To do this, we copy the current network weights to obtain an action-value function $\hat{Q}$ to generate target values y_j. By doing this, we add a delay between the time an update to Q is made and the time the update affects the targets y_j, making divergence or oscillations much more unlikely.

The loss function used to update the weights of the parametrized action-value function is:

$$L_i(\theta_i) = E_{s,a,r,s'} \sim U(D)\big[(r + \gamma \max_{a'} Q(s',a';Q_i^- - Q(s,a;\theta_i)^2\big] \tag{5.1.1}$$

Where $U(D)$ means that we take a uniformly random subset of the data set D. θ_i^- are the network parameters used to compute the target value. γ determines the importance of future rewards. Instead of stochastic gradient descent, RMSProp is used here to update the weights.

The deep convolutional neural network gets as input the pixel colors of a game screen and has an output unit for every possible action. As such, we can compute the action-value function for every action when being in a certain state using only one forward pass.

To select an action, we can use the classic action selection policies such as ϵ-greedy and softmax. Using these policies, we can learn off-policy because we don't always choose the action with the highest action-value. The value of respectively ϵ or τ in these mentioned policies can change over time to get closer to on-policy.

All rewards were clipped between -1 and 1 to limit the scale of derivatives and as such the amount of change of the weights. It also allows for using the same learning rate for multiple games.

The resulting pseudo-code can be seen in Algorithm 6.

Algorithm 6: Deep Q-learning with experience replay. Source: Mnih et al., 2013.

```
1  Initialize replay memory D to capacity N
2  Initialize action-value function $Q$ with random weights $\theta$
3  Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
4  for episode = 1, M do
5      Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
6      for $t = 1, T$ do
7          With probability $\epsilon$ select a random action $a_t$
8          otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
9          Take action $a_t$ and observe reward $r_t$ and new image $x_{t+1}$
10         $s_{t+1} \leftarrow s_t, a_t, x_{t+1}$
11         Preprocess $\phi_{t+1} \leftarrow \phi(s_{t+1})$
12         Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
13         Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
14         if episode terminates at step $j + 1$ then
15             $y_j \leftarrow r_j$
16         else
17             $y_j \leftarrow r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-)$
18         end
19         Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
             network parameters $\theta$
20         Every $C$ steps reset $\hat{Q} \leftarrow Q$
21     end
22 end
```

5.2 Continuous control with deep reinforcement learning

It is also possible to use an Actor-critic algorithm along with a replay buffer like in the DQN algorithm (Mnih et al., 2015). This idea is presented by Lillicrap et al., 2015

For the critic, a function approximator with the following loss is used:

$$L(\theta^Q) = E_{s_t \sim \rho^\beta, a_t \sim \beta, r_t \sim E}\left[(Q(s_t, a_t|\theta^Q) - y_t)^2\right] \tag{5.2.1}$$

Where

$$y_t = r(s_t, a_t) + \gamma Q(s_t, \mu(s_{t+1})|\theta^Q) \tag{5.2.2}$$

$\mu(s_t|\theta^Q)$ is a function that specifies the policy by mapping states to actions. It can be seen that using the loss function we have a critic that is learned using the Bellman equation, just like in Q-learning.

Instead of just copying the weights to the target value network every C steps, we gradually move the target value network values towards the learned network values: $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$ with $\tau \ll 1$. This has the effect that target values can only change slowly. However, although learning is slowed down, it can help to avoid divergence.

Most of the games have different scales for their state values. Possible solutions are to change the hyperparameters for each game individually or to scale the states manually. However, it is also possible to use a more general solution called batch normalization. Here, we scale normalize each dimension across the samples in a minibatch in order to have unit mean and variance. This minimizes the covariance shift during training. To have more exploration, noise is added to the actor policy. This noise is generated using the Ornstein-Uhlenbeck process.

5.3 Asynchronous Methods for Deep Reinforcement Learning

In Mnih et al., 2016 experience replay isn't used because of memory and computation usage and because it is off-policy, which means that it learns from data generated by a policy that may already be outdated. Instead, we use multiple agents (that use well-known reinforcement algorithms) that run in parallel but each running on a separate instance of the same type of environment. These all run on a multi-core CPU (i.e. on only one computer). Each of the agents use the same parameters θ, but by using certain exploration policies, they are able to each explore a possibly different part of the state space. It is possible to use a different exploration policy for each agent as to maximize the exploration of different states. Each agent also calculates a gradient w.r.t. the parameters θ depending on its experience. These gradients are accumulated and after a certain number of steps they are applied to the parameters θ. Because they are global, this update has an effect on all agents. After an amount (possibly the same amount as for the gradient update) of steps, the target network parameters θ^- can also be updated using the current parameters θ.
In n-step Q-learning and advantage actor-critic, a forward-view is used instead of the more commonly used backward view (which uses eligibility traces). Forward view was found to be easier when training neural networks with momentum-based methods and backpropagation through time. To do this, we first compute a certain number of steps (or until the episode is over). Then, the gradients are computed for each state-action pair encountered since the last update. Each n-step update uses the longest possible n-step return resulting in a one-step update for the last state, a two-step update for the second last state, and so on for a total of up to the previously determined number of maximum allowed steps. These accumulated gradients are then immediately applied to θ. This results in the pseudo-code shown in Algorithm 7.

Algorithm 7: Asynchronous Advantage Actor Critic (A3C). Source: Mnih et al., 2016.

```
1  // Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter
     $T = 0$
2  // Assume thread-specific parameter vectors $\theta'$ and $\theta'_v$
3  Initialize thread step counter $t \leftarrow 1$
4  repeat
5      Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
6      Synchronize agent-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
7      $t_{start} \leftarrow t$
8      Initialize state $s_t$
9      repeat
10         Perform $a_t$ according to policy $\pi(a_t|s_t;\theta')$
11         Receive reward $r_t$ and new state $s_{t+1}$
12         $t \leftarrow t + 1$
13         $T \leftarrow T + 1$
14     until terminal $s_t$ or $t - t_{start} == t_{max}$
15     $R = \begin{cases} 0 & \text{if } s_t \text{ is terminal} \\ V(s_t, \theta'_v) & \text{otherwise // Bootstrap from last state} \end{cases}$
16     for $i \in \{t-1, \ldots, t_{start}\}$ do
17         Accumulate gradients w.r.t. $\theta'$: $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
18         Accumulate gradients w.r.t. $\theta'_v$: $d\theta_v \leftarrow d\theta_v + \partial\,(R - V(s_i;\theta'_v))^2 / \partial\theta'_v$
19     end
20     Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$
21 until $T > T_{max}$
```

Chapter 6

Transfer learning

Transfer learning involves the use of experience gained when learning one or more tasks, to improve the performance on a different but related task. Each task is represented by a Markov Decision Process (MDP).
Here, we discuss a framework for transfer methods that can be used for reinforcement learning, closely following Taylor and Stone, 2009.

As already said, the transfer of knowledge always happens between one or more source tasks and a target task. First, an appropriate set of tasks must be selected. Afterwards, the transfer learning algorithm must learn how these source tasks are related to the target task. Then, the appropriate knowledge can be transferred.

6.1 Transfer learning dimensions

To be able to transfer knowledge, some assumptions must be made about differences between the source tasks and the target task. This can be for example in the underlying dynamics of the environment, which can make the task harder or easier to solve, or different sets of possible actions at certain states. These differences define between which type of source and target tasks knowledge can be transferred. The differences between the source task(s) and target task can also make the knowledge transfer easier or harder, requiring the appropriate guidance by a human or a method that can overcome these differences in case of a fully autonomous transfer learner.
Multi-task learning is a special kind of task learning where the problems for the source and target tasks are drawn from the same distribution instead of having arbitrary source and target tasks. More specifically, the transition function is drawn from a fixed distribution of functions. For the mountain car environment, this may mean for example that the motor of the car differs in power in different tasks.

As was stated, first the set of source tasks needs to be selected. Again, this can be done by a human in case of a human-guided scenario. However, the selection may also be done by the agent itself. For example, it can learn multiple source tasks and then use them all for transfer. Another possibility is to select the source tasks that are the most relevant and lead to highest performance for the target task. The agent may also just aim to avoid negative transfer, such that the specific selection of source tasks does not to worsen learning performance for the target task. The agent could also modify the source task(s) such that the transferred knowledge is the most useful in the target task.

Instead of just knowing that tasks are related, many methods also need to know how tasks are related, using task mappings. This is necessary to make the knowledge gained on the source tasks useful for the target task. Tasks may differ in state and action variables and the in the semantic meaning of them. In the mountain car environment, one can have a task where the goal is on the opposite side. As such, taking the action `Left` has a different meaning for these tasks. Actions in the two tasks must be mapped such that their effects are similar.
Again, these mappings can be provided by a human or they can be learned by the agent. Note that these mappings may be partial and not every action in the source task is mapped to an action in the target task or vice-versa. For the state space, it is also possible to map the states themselves instead of the state variables.
In multi-task learning, states and variables are the same and have the same meaning. Because of this, no task mappings are necessary in multi-task learning.

Task learning methods can also differ in the type of knowledge that is transferred between source and target tasks. This knowledge can be for example an action-value function, transitions or policy gradient parameters. For tasks that are closely related, detailed knowledge may be useful. Otherwise, high-level information may result in a better learning performance. The type of knowledge that is transferred can also depend on the type of source and target tasks and on the task mappings.

Last, the task learning method may also restrict which reinforcement learning algorithms that can be used. It is possible for example that only a class of reinforcement learning algorithms or only one specific reinforcement learning algorithm can be used with the task learning method or that the algorithm is the same for both the source and target tasks. Ideally, the reinforcement learning algorithms can be chosen freely and may be selected based on the characteristics of the task at hand.

6.2 Metrics

Several metrics exist to evaluate the learning performance and solution quality of transfer learning algorithms. Generally, one metric does not give a complete representation of the overall performance of the algorithm. Because of this, often multiple metrics are used.
Here we will list the most popular ones:

- *Jumpstart*: This is the initial improvement that the target task has over an algorithm that doesn't use knowledge transfer. However, little to no learning has occurred yet. Because of this, learning performance can't be measured. The metric also does not give an indication about the final performance, i.e. the performance after having learned.

- *Asymptotic performance*: This is the opposite of *jumpstart* performance and measures the performance improvement after having learned the target task or during the last time steps of the algorithm. However, this is hard to measure because one has to know when the task learning algorithm converged. Furthermore, the task learning algorithm and the algorithm that doesn't use knowledge transfer may require a different training time in order to converge.

- *Total reward*: This is the total reward that the algorithm has accumulated during learning, i.e. the area under the learning curve. In case of improvement, this area is bigger than when transfer learning isn't used. This can be achieved when the transfer learning algorithm has a higher learning rate. However, convergence is again an issue here. An algorithm that learns slower and thus takes more time to converge may accumulate a higher total reward than a faster algorithm, although the latter may even reach a higher performance. This metric is only useful for task that always have the same duration.

- *Transfer ratio*: This is the ratio of the total reward improvement that the transfer learning algorithm has over the other algorithm. Because it uses the *total reward* metric, it suffers from the same issues. Furthermore, it is also influenced by the reward structure. For example, an agent always receiving a reward of +1 at the end of the episode may result in a different ratio.

- *Time to threshold*: This measures the time needed to reach a pre-defined performance. A transfer learning algorithm may need less time to reach this threshold. The threshold needs to be defined manually and depends on the task domain and learning method.

A graph with 3 of the 5 metrics is shown in Figure 6.1. Instead of comparing the transfer learning algorithm with a algorithm that doesn't use knowledge transfer, it is also possible to compare the algorithm with the performance of humans (by averaging their performances). However, metrics must be chosen carefully such that they don't favor the algorithm or the human and they can be abused.

6.3 Related work

Transfer learning methods for reinforcement learning can first be divided into 2 groups: methods that use task mappings and methods that do not. We will only discuss methods that do not use task mappings as our approach also does not use task mappings.

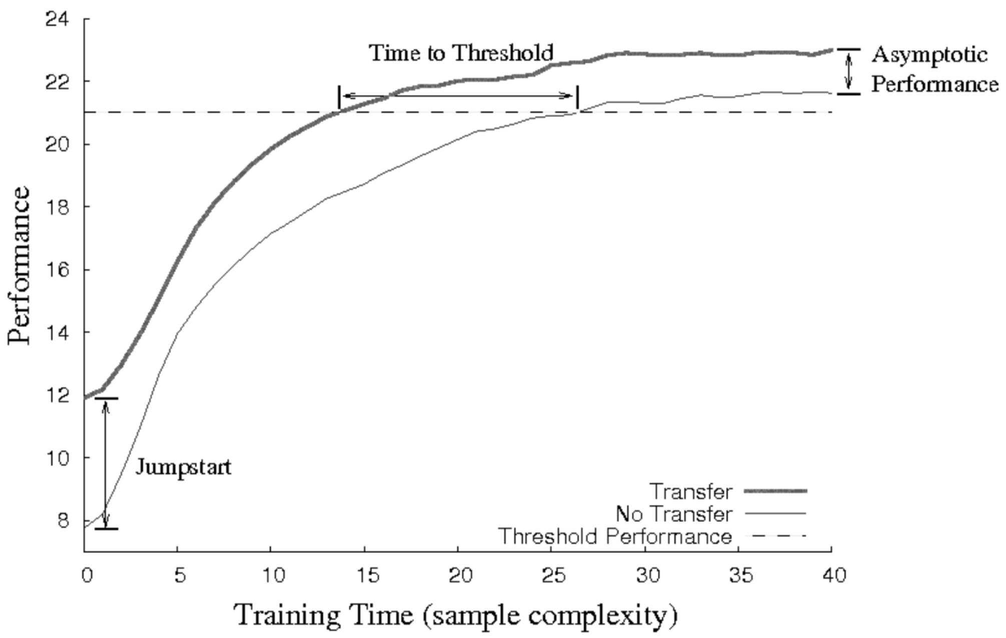

Figure 6.1: A graph comparing the *jumpstart*, *time to threshold* and *asymptotic performance* metrics between the algorithm that uses knowledge transfer and the one that doesn't. Note that here the transfer learning algorithm performs better on all three metrics. It can also be seen that the total reward is higher. Source: Taylor and Stone, 2009.

In one of the earliest works of transfer learning used for reinforcement learning (Selfridge, Sutton, & Barto, 1985), the transition function was gradually changed as to make the task harder. A cart pole task was used where the pole was long and light at first, but was made shorter and heavier over time. The total time spent learning on the sequence of gradually modified tasks was shorter than when trying to solve the hardest task directly.

Later work was generally focused on transferring from one source task to a target task, while in recent research, most of the times a more general scenario with possibly multiple sources is considered. The approaches that we will further discuss mostly focus on transferring from multiple sources instead of just one. We do this because our approach also allows the use of multiple source tasks. Furthermore, the discussed algorithms that can use multiple source tasks generally also allow just one source task from which to transfer knowledge.

One approach, presented in Lazaric, Restelli, and Bonarini, 2008, is to transfer transitions (also called *instances* or *samples*) gathered from the source tasks to the target task. However, transitions may not be useful if the target tasks differs too much from the source tasks. To solve this, transitions from source tasks are chosen based on the similarity of these tasks to the target task. After having trained on the source tasks and collected transitions, we also collect a few transitions from the target task. The similarity between a source task and the target task is then the probability that the transitions of the target task were generated by an approximated model of the source task.

Ammar and Eaton, 2014 also focus on the problem of selecting the appropriate source tasks from which to transfer knowledge. Here, however, a Restricted Boltzmann Machine (Smolensky, 1986) is used in order to generate a model of a source tasks that yields features that are relevant for characterizing the task. The model, generated using transitions of the source task, then tries to reconstruct samples of the target task. The similarity of a source and target task can then be assessed using the difference of the reconstructed transitions and the real transitions of the target task. Note however that the state and action spaces of the source tasks and target task needs to be the same.

Another possibility is to transfer a representation. In this case, characteristics are inferred from multiple source tasks. In Bernstein, 1999, policies are averaged and can be applied for n time steps on all states. This combination of policy, time steps to execute and states is called an option. The reasoning is that actions that are used a lot in a state in source tasks may also be useful in the target task. In Perkins, Precup, et al., 1999, these options are provided on beforehand. The agent then learns a single action-value function over these options using all the source tasks. These options along with the action-value function can then be transferred and used for the target task.

Instead of using a single policy or action-value function, one can also collect a library of policies and select one to use probabilistically, depending on the expected reward. This approach is presented in Fernández and Veloso, 2006; Fernández and Veloso, 2013. At every time step, the algorithm can choose to use one of the source task policies, use the current best target task policy or to randomly explore. As the probabilities depend on the gained rewards, after some time more useful policies are exploited more often. This method only works however for tasks where only the goal state in a maze is different.

Unlike Tanaka and Yamamura, 2003, action-value functions are transferred instead of policies and statistics about them are used. More specifically, the average and standard deviation of the value each state-action pair is calculated. In the target task, the action-value function for each state-action pair is then instantiated to the average for that pair for the source tasks. The standard deviation is used in order to prioritize updates for state-action pairs that are far from the average. Besides this, states-action pairs that fluctuate often within an episode are also prioritized.

Foster and Dayan, 2002 try to extract sub-tasks across multiple source tasks. Optimal policies are then learned for each of these sub-tasks and pieced together. In this case, the environment was a maze and tasks differ in their goal in the maze. Value functions can then be learned on parts of the maze. Less learning for the target task is then required because most of the already learned sub-tasks can also be used for the target task.

Most methods focus on problems with a discrete state and action space. However, other methods exist that can be applied on problems with a continuous state and action space. Here, function approximation is always required. Walsh, Li, and Littman, 2006 group states encountered in the source tasks and treat them as being one and the same state. This abstraction along with the value function learned on this abstraction can then be transferred and used for learning the target task.
Lazaric, 2008 also groups states of the source tasks, but does this by adjusting parameters of a function approximator to build features. Only a small set of features are searched while still being able to learn useful value functions. The learning process for the tasks is executed in parallel. Again, after learning the source tasks, the parameters of the function approximator and the value functions are transferred.

2 other approaches use a hierarchical Bayesian model. They use this to find parameters that make up the dynamics and reward function of a problem. In Sunmola and Wyatt, 2006, transitions of source tasks are used as a prior to find parameters for models. After getting transitions from the target task, the most probable model for the target task transitions is transferred and used.
Wilson, Fern, Ray, and Tadepalli, 2007 use a similar approach, but they don't make a distinction between source and target tasks. Instead, problems (MDPs) are executed sequentially and models are built using an already acquired set of transitions and models of previous tasks.

In Isele and Eaton, 2016, a multi-task learning method is presented that learns tasks simultaneously by using a predefined description for each task. When a new task arrives along with its features, obtained from its task descriptor, a policy can be generated that immediately results in a good performance, even though the algorithm was not applied on that task before and the task descriptor, task order and task distribution was not known on beforehand. This is called *zero-shot learning.* As such, knowledge transfer can take place without the need of training data to identify relationships across tasks. For each subsequent task, the policy and feature vector are iteratively improved.
First, it is assumed that the policy parameters for every task t can be factorized using a shared knowledge base $L \in \mathbb{R}^{d \times k}$: $\theta^{(t)} = Ls^{(t)}$. $s^{(t)} \in \mathbb{R}^{(k)}$ are the sparse coefficients over the basis, i.e. the latent representation of the policy's parameters for task t. L is able to capture relationships among policies as this is used to compute every policy, whereas there is a sparse representation for computed for each task separately.
The features of the task are obtained by (possibly non-linear) feature extractors applied on the descriptor of the task: $\phi(m^{(t)}) \in \mathbb{R}^{d_m}$. These features can also be linearly factorized using another shared knowledge base $D \in \mathbb{R}^{d_m \times k}$: $\phi(m^{(t)}) = Ds^{(t)}$, where the same sparse representation is used as the one used to compute the policy. Both knowledge bases provide information about the task and because of this they share the same coefficient vectors S. We then used coupled dictionary optimization techniques from the sparse coding literature to optimize these dictionaries. They are updated iteratively

based on trajectories samples from the current task.
When a new task arrives, with its task descriptor, we search for the coefficients $s^{(t_{new})}$ that minimize the difference between the extracted features and the reconstruction of it using the shared knowledge base D. Using these coefficients, we can compute the policy parameters using $\theta = Ls^{(t_{new})}$. Afterwards, we can iterate again to improve the sparse representation s and the knowledge bases.

Together with the interest in deep reinforcement learning (Mnih et al., 2015; Mnih et al., 2016), the interest in its application to transfer learning also grows. In Parisotto, Ba, and Salakhutdinov, 2015, the algorithm, Actor-Mimic Network (AMN), learns the Q function for source tasks from an expert for each specific task (in this paper a DQN that was trained until convergence). Note that only one set of parameters (and thus one neural network) is used to learn on all the source tasks. As can be seen, this is supervised learning (more specifically regression) with the output of the expert's network as target value. As input, both the expert and the AMN can be used to sample trajectories. Besides mimicking using the expert's output, the AMN also tries to mimic the hidden layer activations of the expert's network. This is done by adding a loss in function of the difference between the hidden layer activations of the expert network and the activations of the AMN. Intuitively, this gives insight to the AMN, also called to student, why it should act in a specific way, in addition to telling how it should act. After learning for a pre-determined number of frames on each task, the weights of the AMN are transferred to a DQN and the algorithm trains on an unseen target task. Note that each task is a different kind of task here (i.e. the state space, action space, transition function and reward function can be different). However, the algorithm can also be applied to tasks with the same state and action space.

In contrast to the previous algorithm, Progressive Neural Networks (Rusu et al., 2016) use A3C to learn tasks. However, the focus here is on avoiding catastrophic forgetting, which causes previously learned weights to be overwritten when there is a different distribution in the data. In this case, this means that we are learning a different task. To accomplish this, a different set of artificial neural network weights is used for each task. Knowledge is transferred using lateral connections from neurons of artificial neural networks used for previously learned tasks.
After a task is learned, the weights of the used artificial neural networks are kept fixed. When a new task has to be learned, a new artificial neural network is instantiated, with each neuron being both the result of its own knowledge and that of previous tasks:

$$h_i^{(k)} = f\left(W_i^{(k)} h_{i-1}^{(k)} + \sum_{j<k} U_i^{(k:j)} h_{i-1}^{(j)}\right) \tag{6.3.1}$$

Where $h_i^{(k)}$ is the output for layer i of task k, f is an element-wise activation function, $W_i^{(k)}$ is the weight vector for layer i of task k and $U_i^{(k:j)}$ are the lateral connections from layer $i-1$ of task j to layer i of task k.
The algorithm learns both its own weights and its lateral connections from the previous

tasks. Thus, it can decide which knowledge from previous tasks is useful. However, it is not necessary to use this knowledge from previous tasks, as they may not be similar enough for the current task. The task also cannot be influenced by future tasks, as it has no lateral connections from them.
A limitation of this algorithm is that each task has its own set of weights, of which is reported that only a fraction is utilized.

Chapter 7

Proposed algorithm

Our aim is to learn from multiple source tasks using shared knowledge in order to have a better performance on a target task than when we wouldn't train on source tasks. While a version of the algorithm was implemented where the source tasks are executed sequentially for each episode, the focus is on a version where source tasks are learned simultaneously. More specifically, we combine the A3C algorithm (Mnih et al., 2016) with the transfer learning algorithm in Isele and Eaton, 2016. This is done by executing A3C using tasks defined by different environment parameters instead of exactly the same ones and training them using both shared knowledge and knowledge that is specific to the task.

Our approach only uses a latent basis over the policy space instead of using one over the policy space and another one over the descriptor space like in Isele and Eaton, 2016. For a task t, we can reconstruct the policy using the shared basis L and the sparse coefficients over this basis for that specific task $s^{(t)}$: $\theta^{(t)} = Ls^{(t)}$, with $\theta^{(t)} \in \mathbb{R}^d$, $L \in \mathbb{R}^{d \times k}$ and $s^{(t)} \in \mathbb{R}^k$. The policy can then be combined with the state or features extracted from it by taking the weighted sum and passing it through a non-linearity, in this case the softmax function described in Section 2.2.5. This results in a probability for each action.
The architecture of the network used for one task is visualized in Figure 7.1.

The source tasks can be learned in two ways, either sequentially or in parallel. In the sequential way, we collect trajectories and calculate the gradients for every task after another. These gradients include both those for the sparse representation and for the knowledge base. They are calculated for every episode of a task like in the *REINFORCE* algorithm (explained in Section 3.6), i.e. with likelihood ratios using discounted rewards as a sample for the Q values of the policy. After evaluating every task, we sum and apply all of their gradients. The learning process for the source tasks stops when a certain number of updates have been applied.
In the parallel way, all source tasks are learned at the same time, each executing a certain number of updates. Trajectories are collected for each task continuously and its

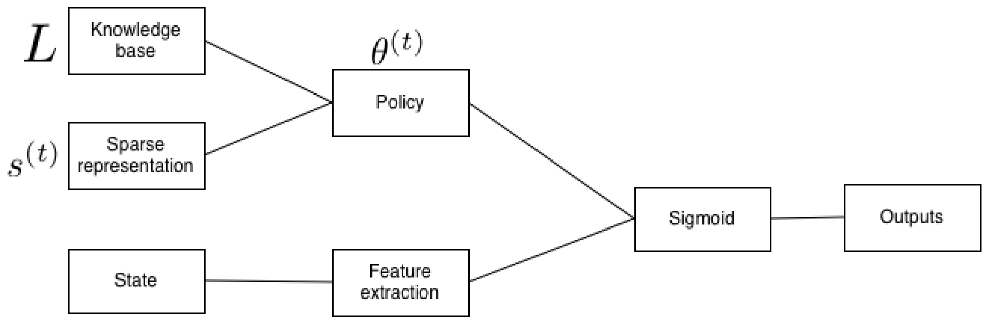

Figure 7.1: Artificial neural network architecture used in our approach. The sparse representation $\theta^{(t)}$ is different for every task t, while the same knowledge base is used for every task.

gradients are applied immediately after they are calculated. This means that they are not first summed up like in the sequential method.
The resulting pseudo code for the source tasks learned using the parallel method can be seen in Algorithm 8.

Algorithm 8: Asynchronous knowledge transfer agent for a source task.

```
1  // Assume global knowledge base $L$ and global shared counter $T \leftarrow 0$
2  Initialize thread-specific parameter vector $\theta^{(t)}$
3  Initialize thread step counter $t \leftarrow 1$
4  repeat
5      $\theta^{(t)} \leftarrow Ls^{(t)}$
6      Reset gradients: $d\theta^{(t)} \leftarrow 0$
7      $t_{start} \leftarrow t$
8      Initialize state $s_t$
9      repeat
10         Perform $a_t$ according to policy $\pi(a_t|s_t;\theta^{(t)})$
11         Receive reward $r_t$ and new state $s_{t+1}$
12         $t \leftarrow t+1$
13     until terminal $s_t$ or $t - t_{start} = t_{max}$
14     for $i \in \{t-1,\dots,t_{start}\}$ do
15         Accumulate gradients w.r.t. $\theta^{(t)}$: $d\theta^{(t)} \leftarrow d\theta^{(t)} + \alpha\nabla_{\theta^{(t)}} \log \pi_{\theta^{(t)}}(s_i,a_i)v_i$
16     end
17     Perform asynchronous update of $\theta^{(t)}$ using $d\theta^{(t)}$
18     $T \leftarrow T+1$
19 until $T > T_{max}$
```

After learning the source tasks, the knowledge base that they jointly learned is transferred to the target task. This task then separately learns and executes updates for a number of episodes. The pseudo-code for the target task is shown in Algorithm 9.

Algorithm 9: Knowledge transfer agent for the target task.

```
1  // Assume global knowledge base $L$
2  Initialize task-specific sparse representation $s^{(t)}$
3  Initialize thread step counter $t \leftarrow 1$
4  $T \leftarrow 0$
5  repeat
6      $t_{start} \leftarrow t$
7      Initialize state $s_t$
8      repeat
9          Perform $a_t$ according to policy $\pi(a_t|s_t; Ls^{(t)})$
10         Receive reward $r_t$ and new state $s_{t+1}$
11         $t \leftarrow t + 1$
12     until terminal $s_t$ or $t - t_{start} = t_{max}$
13     for $i \in \{t-1, \dots, t_{start}\}$ do
14         Accumulate gradients w.r.t. $s^{(t)}$: $ds^{(t)} \leftarrow ds^{(t)} + \alpha \nabla_{s^{(t)}} \log \pi_{s^{(t)}}(s_i, a_i) v_i$
15     end
16     Perform update of $s^{(t)}$ using $ds^{(t)}$
17     $T \leftarrow T + 1$
18 until $T > T_{max}$
```

Chapter 8

Experimental setup

Experiments were executed on variations of the *cart-pole* environment and on variations of the *acrobot* environment. For executing algorithms on both environments, a framework by Brockman et al., 2016 was used.
After describing these environments, we describe the methodology of our experiments and the results.

8.1 Cart-pole environment

In the 2-dimensional *cart-pole* environment (Barto, Sutton, & Anderson, 1983), a pole is placed vertically on a cart. The goal in this environment is to keep the pole balanced vertically (i.e. keep the angle of the pole between thresholds) and to keep the cart between 2 borders. An agent can either move left or right. It can't stay at its current position. The state is defined by 4 continuous-valued attributes: the position of the cart, the velocity of the cart, the angle of the pole and the angular velocity of the pole. A discrete value of 1 is given as a reward each time the pole is balanced and the cart is between bounds. The episode ends either when these requirements are not fulfilled anymore or 200 steps have been executed. The environment is visualized in Figure 8.1.

8.2 Acrobot environment

The goal in the *acrobot* environment, visualized in Figure 8.2, is to swing up the tip of 2 joined arms above a certain point (Spong, 1995). This can be done by applying a force on the joint between the 2 arms. However, this force is not enough to fulfill the goal immediately. Instead, the actuator must apply force to the left and to the right to build up enough speed to get above the horizontal threshold. The state consists of the angle and angular velocity of both arms. One can either move the joint clockwise, counter-clockwise or do nothing. An episode stops either when the tip of the outermost arm is above the threshold or when 500 steps have been executed in the episode. At

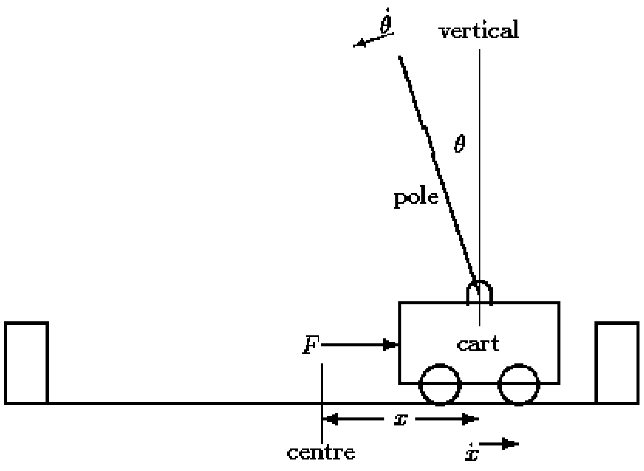

Figure 8.1: Visualization of the *cart-pole* environment. F is the force applied to the cart when taking an action. x is the distance of the cart from the center and $\dot{x}$ is its velocity. θ is the angle of the pole and $\dot{\theta}$ refers to its angular velocity. Source: Grant, 1990.

each step, a reward of -1 is given. The goal is to maximize the reward and as such to minimize the amount of steps necessary to reach the threshold.

8.3 Methodology

Several experiments were executed with our transfer learning algorithm, using varying types of transferred knowledge and artificial neural networks and a different number of source tasks. However, the structure of the algorithm is always the same.
In some experiments, the performance of our algorithm is also compared to the performance of the *REINFORCE* algorithm described in Section 3.6. When necessary, changes to this algorithm are mentioned.
The used values for hyperparameters for both algorithms are described in Appendix A.2.

For an experiment with our algorithm, first a number of environments are randomly generated. We try to learn with either 5 or 10 environments and thus source tasks. The environment for each task can differ for a predefined number of parameters specific to the task, of which the values can each be in a certain range. For example for a *cart-pole* task, these are the mass of the cart, the mass of the pole and the length of the pole. The parameters and the allowed ranges for their values are described for each environment in Appendix A.3.
Then a number of source tasks are learned that can update both the shared knowledge base and their own sparse representation. After a number of epochs, i.e. updates to these variables, we stop with learning these tasks. Instead, we learn to solve a new task, the target task. In some experiments, the sparse representation for the target task is initialized using the sparse representation of a randomly chosen source task. In others, the

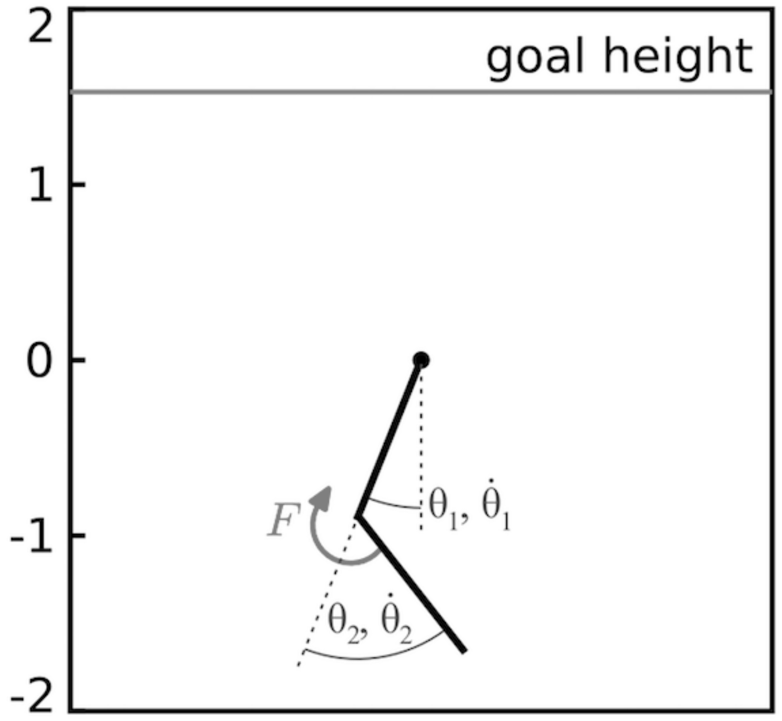

Figure 8.2: Visualization of the *acrobot* environment. θ_1 and $\dot{\theta_1}$ are respectively the angle and angular velocity of the arm attached to the central fixed point. θ_2 and $\dot{\theta_2}$ are respectively the angle and angular velocity of the outermost arm. Source: Frémaux, Sprekeler, and Gerstner, 2013.

sparse representation is randomly initialized, as explained in Appendix A.2. However, for the target task only the sparse representation can be updated and not the shared knowledge base.

Other experiments involve the *REINFORCE* algorithm and was used as comparison with the results of the other experiments. Here, we first run the algorithm on one source task and then transfer the learned knowledge to the target task. In practice, this means that we start learning on the target task with the weights learned using the source task. The algorithm can still learn adapt all the weights using experience gained on the target task. No sparseness on the weights was enforced.

Each experiment was repeated 100 times, each time using a different set of environments and thus problems to solve. Afterwards the rewards are averaged over all the runs.
Trajectories for both sets of tasks contained of maximally 200 or 500 steps depending on the type of environment. This could be less in case the environment was in an end state. In case of the *cart-pole* environment, this can mean for example that the cart tried to

cross the left or right border or that the pole fell down.
Hyperparameters, such as the learning rate of neural networks, were chosen manually.

8.4 Results

To evaluate our proposed algorithm, we execute multiple experiments. First we compare the sequential and parallel version of our algorithm. We then see if feature extraction can improve the performance. Afterwards, we discuss results of an experiment where we transfer a sparse representation from a random source task to the target task. Last, we evaluate the results of using a single source task.
When discussing the jumpstart and asymptotic performance, we use the mean reward of respectively the first 5 and the last 5 epochs (i.e. updates to the artificial neural network parameters).

8.4.1 Parallel and sequential knowledge transfer

We first compare the performance of the algorithm that learns the source tasks in parallel with the one that learns them sequentially. We do this for the *cart-pole* environment in order to determine which version of the algorithm to use for further experiments.

The performance of both versions of the algorithm and for both the source tasks and target task is shown in Figure 8.3. Here, we see that, on average, the parallel version

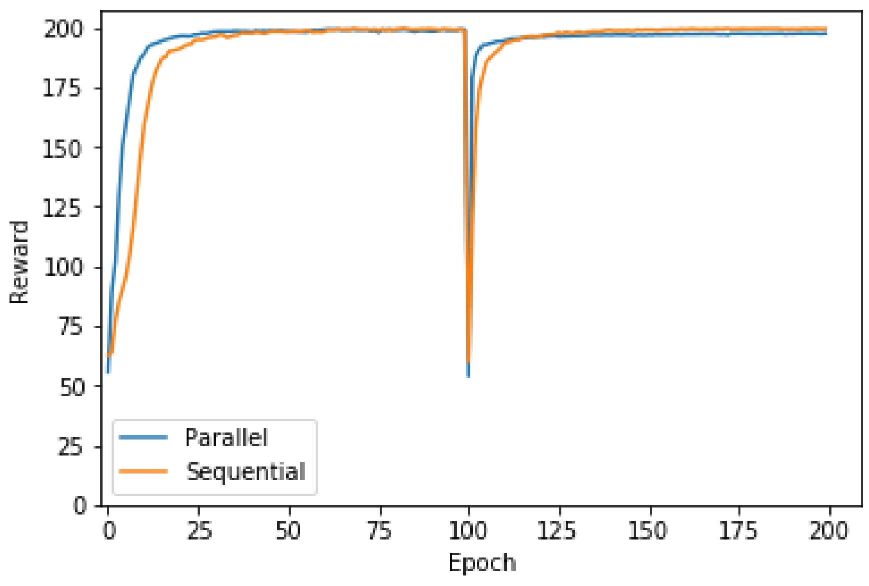

Figure 8.3: Learning curves for the parallel and sequential version of our algorithm applied to the *cart-pole* environment. Until epoch 100, the average performance on the source tasks is shown. Afterwards, the performance of the target task is shown.

of our algorithm converges faster than the sequential version. To find out which version is better, we first compare the median of the area under curve (AUC) over all the runs. For the sequential version, the AUC is 19446.096, while it is 19655.268 for the parallel version. We now use a Wilcoxon rank-sum test with the null-hypothesis that there is no difference. We get $W = 8.244$ and $1.665 * 10^{-16}$. With a significance level of 0.05, we can reject the null-hypothesis and say that the medians of the 2 versions are different. By observing those medians, we can say that the parallel version is better. As it does not first accumulate gradients but applies them as they are computed, it instantly affects other tasks and improves their performance.
For the following experiments, when referring to "our transfer learning algorithm", the parallel version will be used.

8.4.2 Feature extraction

In this experiment, we evaluate the use of feature extraction on the states. We compare the performance of an artificial neural network with a layer of 5 units, one with 10 units and one without feature extraction at all. The learning curves for the performance on the target task are shown in Figure 8.4. Feature extraction does not seem to improve the

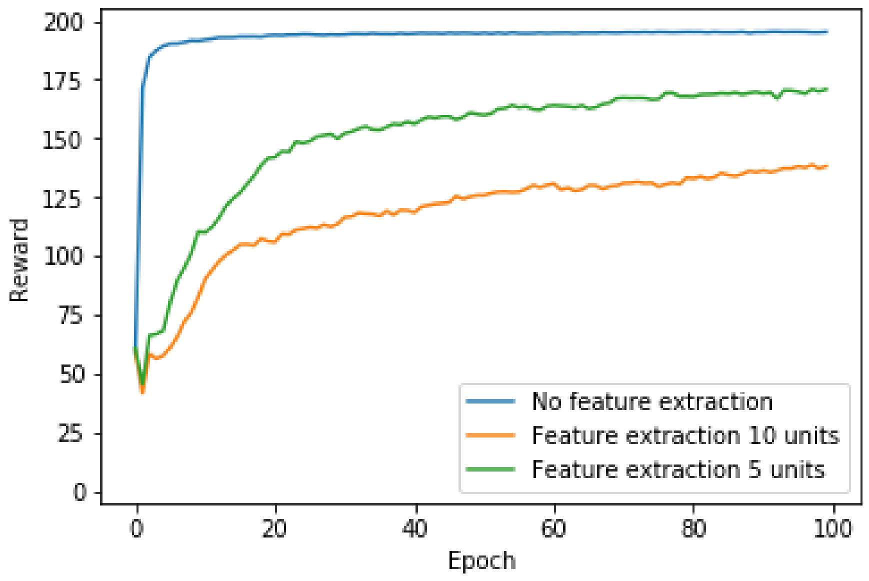

Figure 8.4: Learning curves for the transfer learning algorithm applied to the *cart-pole* environment with no feature extraction in its neural network, one using a layer with 5 units and one using a layer of 10 units.

performance of the algorithm for this environment. It only seems to slow down learning. It is worse when we use more units in the hidden layer used for feature extraction. As more units are added, more weights influence the result and more of these weights must be learned.

It must be noted however that for algorithms working with high-dimensional inputs, feature extraction, using for example deep learning, is necessary to obtain a good performance. In this environment, the state contains values that are useful for solving a task. However, without feature extraction, it is not clear if a certain pixel of an image is important. Feature extraction is able to obtain values relevant for solving the task.

For the following experiments, we will focus on the different algorithms and different ways of transferring knowledge instead of the artificial neural network architecture.

8.4.3 Usage of a different amount of source tasks

We now explore the differences between learning directly on a target task and first learning on source tasks (i.e. using our transfer learning algorithm), using the metrics that we defined in Section 6.2. We consider both 5 and 10 source tasks. The algorithms using 5 source tasks and 10 source tasks are referred to as respectively *TLA 5* and *TLA 10*. For the regular algorithm, we use the *REINFORCE* that was discussed in Section 3.6.

8.4.3.1 Cart-pole

The learning curve of both algorithms on the target task is visualized in Figure 8.5. It

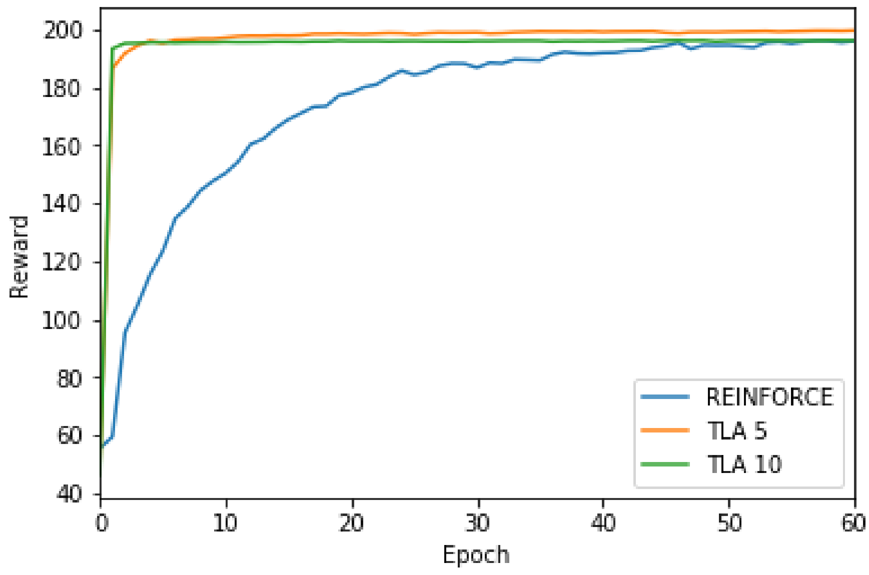

Figure 8.5: Learning curves for the *cart-pole* environment of the *REINFORCE* algorithm and our transfer learning algorithm (*TLA*) with 5 source tasks (shown as *TLA 5*) and 10 source tasks (shown as *TLA 10*). The learning curves are only shown until epoch 60 in order to better show the initial performance of the algorithms. All the algorithms converged and achieved the same rewards after this epoch.

can be seen visually that, on average, in all the configurations the jumpstart performance is about the same. We now compare the means, standard deviations and the medians of the jumpstart performances of the algorithms. This is shown in Table 8.1. It can

Algorithm	Mean	Standard deviation	Median
REINFORCE	86.060	25.330	86.886
TLA 5	158.237	23.153	163.840
TLA 10	**165.120**	21.955	**169.034**

Table 8.1: Mean, standard deviation and median of the jumpstart performances for *REINFORCE*, *TLA 5* and *TLA 10* applied to the cart-pole environment.

be seen that the best jumpstart performance is achieved using *TLA 10*, as it has the highest mean and median and varies the least. Because it learned on more source tasks, it has more samples from the distribution of environment parameters and thus it can cover more kinds of task variations. *REINFORCE* performs the worst, as it must learn without any prior learned knowledge.

We can also see in Figure 8.5 that *REINFORCE* takes longer on average to reach the maximum reward (which is 200 in this environment). As a result, the area under curve for *REINFORCE* is 18170.243 whereas it is 19130.824 and 19329.318 for respectively *TLA 5* and *TLA 10*.
In every configuration, the maximum reward is reached most of the times as the median asymptotic performances for the 3 algorithms are all 200.

8.4.3.2 Acrobot

We now do with the same experiment, but with the *acrobot* environment. As can be seen in Figure 8.6, the average performance of the *REINFORCE* algorithm on the target task is different from *TLA 5* and *TLA 10*.

We can how exactly the behavior of these algorithms differ initially by looking at the jumpstart performances. Statistics of these jumpstart performances are shown in Table 8.2. We see that *TLA 10* performs the best. However, as seen by the standard

Algorithm	Mean	Standard deviation	Median
REINFORCE	−361.209	**159.760**	−490.063
TLA 5	−380.222	160.589	−500.000
TLA 10	**−339.530**	168.845	**−419.571**

Table 8.2: Mean, standard deviation and median of the jumpstart performances for *REINFORCE*, *TLA 5* and *TLA 10* applied to the *acrobot* environment.

deviations, the results can differ considerably. We will see if the differences in medians are significant by using a Wilcoxon rank-sum test with for each pair of algorithms the null-hypothesis that there is no difference between them. The resulting p-values are

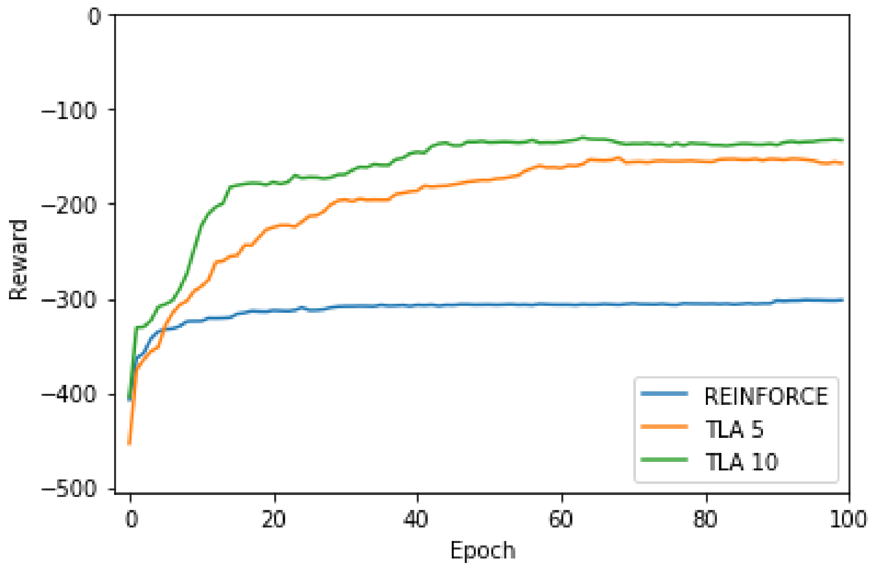

Figure 8.6: Learning curves of the *REINFORCE* algorithm and our transfer learning algorithm (*TLA*) with 5 source tasks (shown as *TLA 5*) and 10 source tasks (shown as *TLA 10*).

shown in Table 8.3. We can say using each time a significance level of 0.05 that there is

Algorithm	*REINFORCE*	*TLA 5*	*TLA 10*
REINFORCE			
TLA 5	0.512		
TLA 10	0.426	0.179	

Table 8.3: P-values for the Wilcoxon rank-sum test of the jumpstart performances using different algorithms, applied to the *acrobot* environment.

no difference between any combination of algorithms.

As we could already see in Figure 8.6, the *REINFORCE* algorithm seems learn slower on average than the other algorithms. This is also visible when looking at the area under curve, which is −30810.360 for *REINFORCE* algorithm and −19437.733 and −16223.803 for respectively *TLA 5* and *TLA 10*.

Last, we compare the asymptotic performances, again with the same test and the null-hypotheses that there are no differences between the asymptotic performances. The resulting p-values are shown in Table 8.4.

With a significance level of 0.05, we can say that there is a difference between *REINFORCE* and *TLA 5* and between *REINFORCE* and *TLA 10*. With the same significance level, we retain the null-hypothesis for *TLA 5* and *TLA 10*. Thus, we can say that us-

Algorithm	*REINFORCE*	*TLA 5*	*TLA 10*
REINFORCE			
TLA 5	$1.034 * 10^{-4}$		
TLA 10	$7.091 * 10^{-6}$	0.449	

Table 8.4: P-values for the Wilcoxon rank-sum test using different algorithms with the *acrobot* environment.

ing knowledge transfer yields a significant difference in the resulting performance, when applied to the *acrobot* environment. To further discuss these differences, we look at the boxplots of the asymptotic performances for the three configurations. These are shown in Figure 8.7. We can see that the median for the *REINFORCE* algorithm is still the

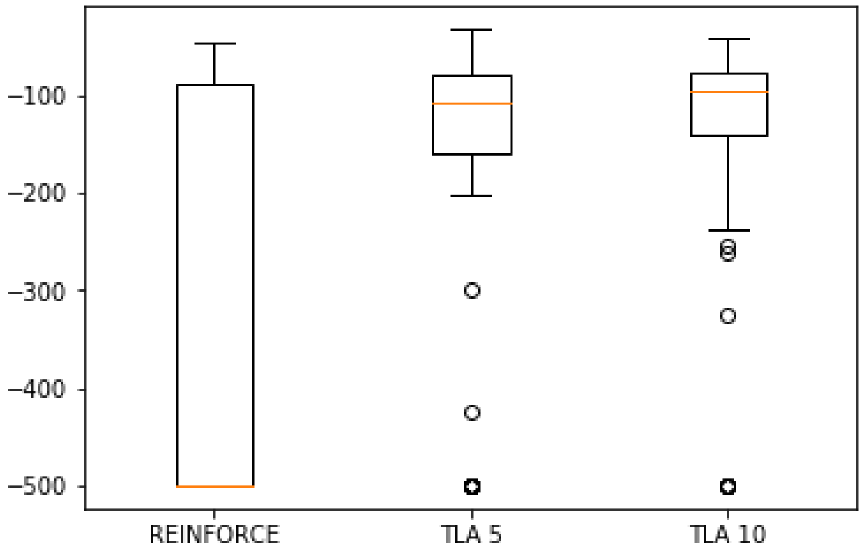

Figure 8.7: Boxplots of the asymptotic performances of *REINFORCE*, *TLA 5* and *TLA 10* using the *acrobot* environment.

lowest possible reward for this environment. This means that in at least half of the cases, the *REINFORCE* algorithm cannot improve and cannot get past the minimum reward. In contrast, the medians of *TLA 5* and *TLA 10* are respectively −105.305 and −98.391. Only some outliers of these algorithms don't get past the minimum reward.

8.4.4 Transfer of sparse representation

We now explore the results of transferring a sparse representation. After learning on the source tasks, we initialize the sparse representation weights of the target task with the sparse representation of a randomly chosen source task. The algorithm can then change this sparse representation to improve its performance on the target task. We compare the results to those when not transferring the sparse representation, both for the cart-pole and the *acrobot* environment.

8.4.4.1 Cart-pole

To see how both version of the algorithm differ on average in terms of learning performance, we take a look at the learning curves, which are shown in Figure 8.8. It can be

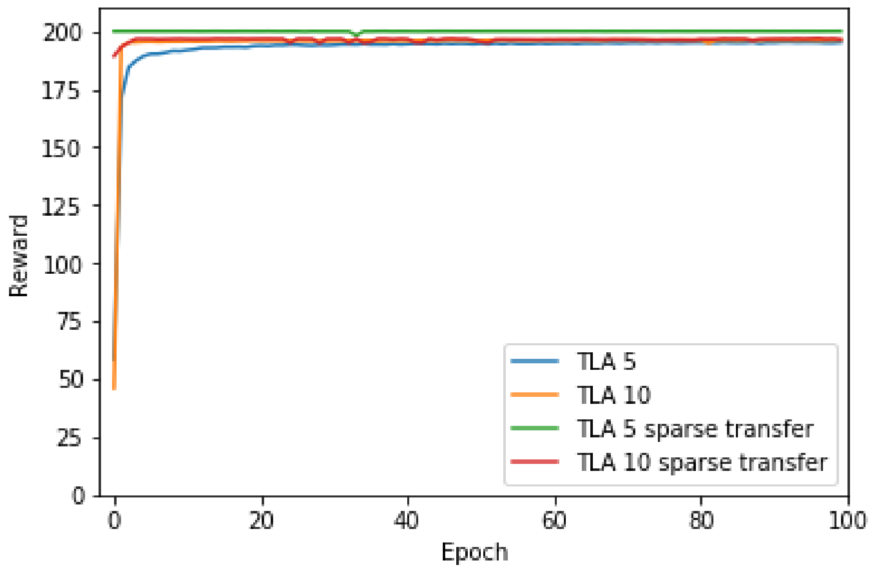

Figure 8.8: Learning curves for the *Cart-pole* environment of our transfer learning algorithm (*TLA*) with 5 source tasks (shown as *TLA 5*) and 10 source tasks (shown as *TLA 10*), both with and without sparse representation transfer.

seen that the versions using sparse representation transfer need little to no update to reach the maximum reward. However, all versions converge to the maximum as all the medians of the asymptotic performances are 200.

We now compare the jumpstart performances of these algorithms using the mean, standard deviation and median, shown in Table 8.5. As shown, the versions using

Algorithm	Sparse repr. transfer	Mean	Standard deviation	Median
TLA 5	No	158.237	23.153	163.840
TLA 10	No	165.120	21.955	169.034
TLA 5	Yes	192.613	28.179	**200.000**
TLA 10	Yes	**196.125**	21.640	**200.000**

Table 8.5: Mean, standard deviation and median of the jumpstart performances for *TLA 5* and *TLA 10*, with without and with sparse representation transfer, applied to the cart-pole environment.

sparse representation transfer perform better initially. In at least half the runs, the maximum reward is already reached at the start. Just like for the version without sparse representation transfer, in most runs the maximum is maintained or reached at the end.

8.4.4.2 Acrobot

To discuss the differences between the 2 versions for the *acrobot* environment, we also look at the learning curves, which are shown in Figure 8.9. It can be seen that, on

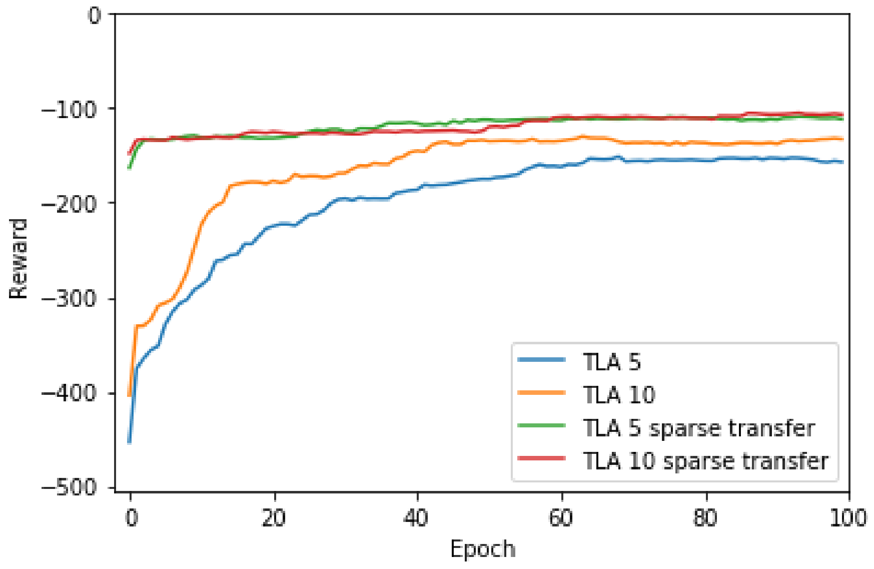

Figure 8.9: Learning curves for the *acrobot* environment of our transfer learning algorithm (*TLA*) with 5 source tasks (shown as *TLA 5*) and 10 source tasks (shown as *TLA 10*), both with and without sparse representation transfer.

average, the version using sparse representation transfer can start with a high reward but improves little afterwards. However, the rewards stay superior to those using the version without sparse representation transfer.

First, we see how the 2 versions perform initially, using the jumpstart performances. Statistics about the jumpstart performances are presented in Table 8.6. Indeed, the versions using sparse representation transfer initially obtain a higher reward and are more stable. This means that the sparse representation of a source task is a better initialization than just using random values.
Next, we use the same types of statistics for the asymptotic performances. These are shown in Table 8.7. Again, the versions with sparse representation transfer perform better, because a higher mean and median reward is achieved and the standard deviations are lower.

Algorithm	Sparse repr. transfer	Mean	Standard deviation	Median
TLA 5	No	−380.222	160.589	−500.000
TLA 10	No	−339.530	168.845	−419.571
TLA 5	Yes	−141.236	101.567	−110.482
TLA 10	Yes	**−136.400**	97.188	**−108.157**

Table 8.6: Mean, standard deviation and median of the jumpstart performances *TLA 5* and *TLA 10* both with and without sparse representation transfer, applied to the *acrobot* environment.

Algorithm	Sparse repr. transfer	Mean	Standard deviation	Median
TLA 5	No	−156.128	137.661	−107.222
TLA 10	No	−132.873	105.777	−96.224
TLA 5	Yes	−110.481	67.384	−93.085
TLA 10	Yes	**−106.626**	64.108	**−91.087**

Table 8.7: Mean, standard deviation and median of the asymptotic performances *TLA 5* and *TLA 10* both with and without sparse representation transfer, applied to the *acrobot* environment.

Using these statistics, we see that for this environment a sparse representation is also a useful and more stable starting point for the target task and also leads to a better performance at the end. We can also notice that the performance is better when using using 10 source tasks instead of 5. To see if this difference is significant, we use a Wilcoxon rank-sum test with the null-hypothesis that there is no difference in asymptotic performances. This results in a p-value of 0.654. Thus, with a significance level of 0.05, we can retain the null-hypothesis and say that the difference is not significant.

8.4.5 REINFORCE using a source and target task

To see if it is really necessary to learn from multiple source tasks that use a shared knowledge base, we apply the *REINFORCE* algorithm first on one source task and then use the learned weights of the artificial neural network as initialization for the network of the target task. Our transfer learning algorithm is used for comparison and first learns on 5 source tasks. It then transfers besides the shared knowledge base also a randomly chosen sparse representation from one of the source tasks, like in Section 8.4.4.
We execute this experiment both for the *cart-pole* and the *acrobot* task.

8.4.5.1 Cart-pole

For the *cart-pole* environment, the learning curves are shown in Figure 8.10. It can be seen that on average the *TLA 5* algorithm learns more quickly on the source tasks. It is also able to retain a better jumpstart performance, with a median of 200 instead of 186.908 for *REINFORCE*. The asymptotic performances seem to be equal and have the

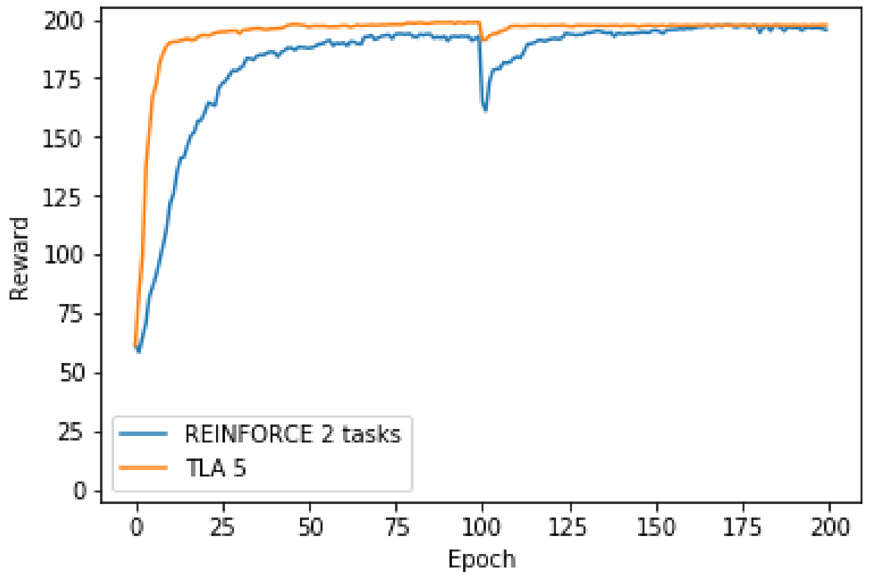

Figure 8.10: *REINFORCE* applied for 100 epochs to a randomly chosen source task of a *cart-pole* environment and afterwards to the target task, using the same network and weight values. The learning curve is compared to the one of the *TLA 5* algorithm that uses sparse representation transfer.

same median of 200.

8.4.5.2 Acrobot

We now do the same experiment for the *acrobot* environment. The resulting learning curve for the source and target task is shown in Figure 8.11. On average, for both algorithms only a small adaption is needed for the target task to reach the same reward as the one with which the source task ended. However, the median jumpstart performance is -500, meaning that in at least half the cases the knowledge of the source task does not improve the initial performance on the target task. In contrast, the median jumpstart performance for *TLA 5* is -110.482. For the target task, the asymptotic performance of *REINFORCE* also does not reach the values from our *TLA 5* algorithm. The *REINFORCE* algorithm has a median asymptotic performance of -203.426 instead of -93.085 for *TLA 5*.

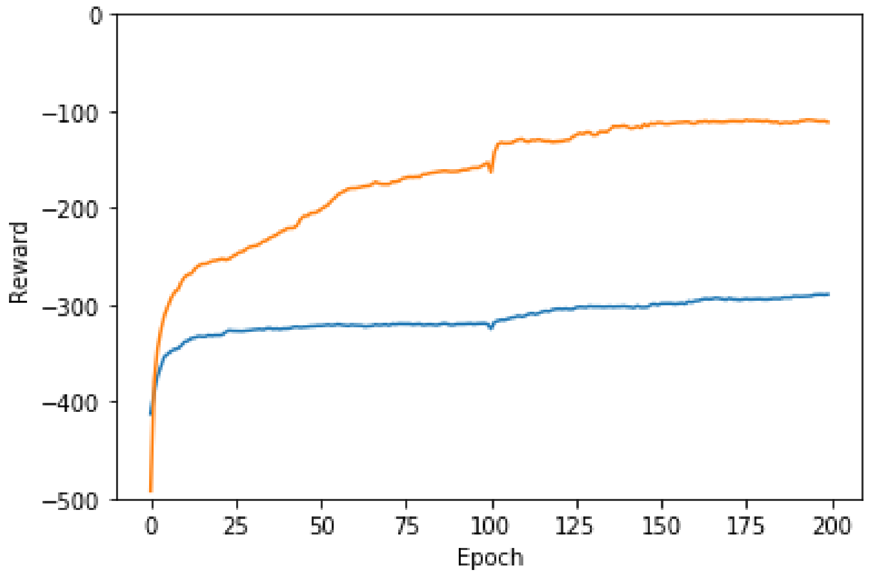

Figure 8.11: *REINFORCE* applied for 100 epochs to a randomly chosen source task of an *acrobot* and afterwards to the target task, using the same network and weight values.

Chapter 9

Conclusion

We presented an algorithm suitable for learning in parallel or sequentially on a set of source tasks. These task share a knowledge base, but they also have their proper sparse representation. The learned knowledge can then be transferred to the target task with the goal of having an increased performance over an algorithm that does not use prior knowledge.

In our experiments, we found that source tasks learned in parallel learned faster than when they were learned sequentially. In the parallel version, changes to the shared knowledge base are applied immediately instead of summing them and applying the changes after all source tasks have been evaluated.

We also discussed to use of feature extraction applied to the input, which is in this case the state of an environment. For the cart-pole environment, no feature extraction was necessary and even slowed down learning. However, environments with a high-dimensional input space generally require feature extraction in order to obtain the relevant aspects of the input.

Our algorithm has better performance on the target task than when just using the *REINFORCE* algorithm on it. Our algorithm learns faster and is able to receive higher rewards. The performance is even better when we transfer the sparse representation from a randomly chosen source task to the target task. The algorithm then only needs to tune the sparse representation for it to work on its own task.

To see if multiple source tasks are really necessary, we compared our algorithm with the *REINFORCE* algorithm where it learns on a single source task and transfers all its knowledge to the target task. Although the asymptotic performance was similar, our algorithm learned better on the source tasks and has a higher jumpstart performance.

We can conclude that it is beneficial to learn on multiple source tasks in parallel and transfer knowledge learned on these tasks to the target task.

Appendices

Appendix A

Experiment details

A.1 Data collection

For each agent, transitions on which to learn were collected in the same way. In each iteration, 5000 transitions have to be collected. To do this, we keep executing actions and saving the transitions until this threshold is reached. When an end state is reached before reaching this threshold, we reset the environment and keep executing actions.

A.2 Artificial neural network parameters

Weights are initialized using values drawn from a truncated normal distribution with mean 0 and standard deviation 0.02. This means that we draw values from a normal distribution with the same mean and standard deviation, except that only values maximally 2 standard deviations away from the mean can be drawn.
To update weights, I used *RMSProp*, described in Section 2.3.4.2. The following hyperparameters were always used for both the sequential and the parallel version of the algorithm and for *REINFORCE*:

- $\epsilon = 10^{-9}$
- $\alpha = 0.05$
- $\gamma = 0.9$

A.3 Environment parameters

Here we describe the range of possible values for each parameter of the *cart-pole* and *acrobot* environments. To generate an environment, each parameters is drawn from a uniform distribution with possible values in the defined range. For the *cart-pole* environment, we have:

Parameter name	**minimum**	**maximum**
Pole length	0.01	5.0
Pole mass	0.01	5.0
Cart mass	0.01	5.0

For the *acrobot* environment, we have:

Parameter name	**minimum**	**maximum**
Length of link 1	0.2	2.0
Length of link 2	0.2	2.0
Mass of link 1	0.2	2.0
Mass of link 2	0.2	2.0

Bibliography

Ammar, H. B. & Eaton, E. (2014). An Automated Measure of MDP Similarity for Transfer in Reinforcement Learning. *Workshops at the Twenty-Eighth AAAI Conference on Artificial Intelligence*, 31–37. Retrieved from http://www.aaai.org/ocs/index.php/WS/AAAIW14/paper/viewPaper/8824

Anderson, C. W. (1989). Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine*, *9*(3), 31–37.

Bach, F. R., Jenatton, R., Mairal, J., & Obozinski, G. (2012). Optimization with sparsity-inducing penalties. *Foundations and Trends in Machine Learning*, *4*(1), 1–106. Retrieved from http://dblp.uni-trier.de/db/journals/ftml/ftml4.html#BachJMO12

Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Trans. Systems, Man, and Cybernetics*, *13*(5), 834–846. Retrieved from http://dblp.uni-trier.de/db/journals/tsmc/tsmc13.html#BartoSA83

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, *5*(2), 157–166.

Bernstein, D. S. (1999). *Reusing Old Policies to Accelerate Learning on New MDPs.*

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI Gym. *arXiv*, 1–4. Retrieved from http://arxiv.org/abs/1606.01540

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1724–1734. doi:10.3115/v1/D14-1179

Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., & LeCun, Y. (2015). The Loss Surfaces of Multilayer Networks. In *Aistats.*

Chung, J., Gülçehre, Ç., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, *abs/1412.3555.* Retrieved from http://dblp.uni-trier.de/db/journals/corr/corr1412.html#ChungGCB14

Clevert, D.-A., Unterthiner, T., & Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). *CoRR*, *abs/1511.07289.* Retrieved from http://dblp.uni-trier.de/db/journals/corr/corr1511.html#ClevertUH15

Cottrell, G. W. (1990). Extracting features from faces using compression networks: Face, identity, emotion and gender recognition using holons. In *Connectionist models: Proceedings of the 1990 summer school* (pp. 328–337).

Demant, C., Garnica, C., & Streicher-Abel, B. (2013). Overview: Classification. In *Industrial image processing: Visual quality control in manufacturing* (pp. 151–172). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-33905-9_6

Fernández, F. & Veloso, M. (2006, September 27). Probabilistic policy reuse in a reinforcement learning agent. In H. Nakashima, M. P. Wellman, G. Weiss, & P. Stone (Eds.), *Aamas* (pp. 720–727). ACM. Retrieved from http://dblp.uni-trier.de/db/conf/atal/aamas2006.html#FernandezV06

Fernández, F. & Veloso, M. M. (2013). Learning domain structure through probabilistic policy reuse in reinforcement learning. *Progress in AI*, *2*(1), 13–27. Retrieved from http://dblp.uni-trier.de/db/journals/pai/pai2.html#FernandezV13

Foster, D. J. & Dayan, P. (2002). Structure in the Space of Value Functions. *Machine Learning*, *49*(2-3), 325–346. Retrieved from http://dblp.uni-trier.de/db/journals/ml/ml49.html#FosterD02

Frémaux, N., Sprekeler, H., & Gerstner, W. (2013). Reinforcement learning using a continuous time actor-critic framework with spiking neurons. *PLOS Computational Biology*, *9*(4), 1–21. doi:10.1371/journal.pcbi.1003024

Grant, S. (1990). Modelling cognitive aspects of complex control tasks. In *Proceedings of the ifip tc13 third interational conference on human-computer interaction* (pp. 1017–1018). North-Holland Publishing Co.

Graves, A. (2013). Generating sequences with recurrent neural networks. *CoRR*, *abs/1308.0850*. Retrieved from http://dblp.uni-trier.de/db/journals/corr/corr1308.html#Graves13

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, *abs/1502.01852*. Retrieved from http://dblp.uni-trier.de/db/journals/corr/corr1502.html#HeZR015

Isele, D. & Eaton, E. (2016). Using Task Features for Zero-Shot Knowledge Transfer in Lifelong Learning. *Ijcai*, 1620–1626.

Kingma, D. P. & Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, *abs/1412.6980*. Retrieved from http://arxiv.org/abs/1412.6980

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger (Eds.), *Nips* (pp. 1106–1114). Retrieved from http://dblp.uni-trier.de/db/conf/nips/nips2012.html#KrizhevskySH12

Lang, K. J., Waibel, A., & Hinton, G. E. (1990). A time-delay neural network architecture for isolated word recognition. *Neural Networks*, *3*(1), 23–43. doi:10.1016/0893-6080(90)90044-L

Lazaric, A. (2008). *Knowledge transfer in reinforcement learning* (Doctoral dissertation, Politecnico di Milano).

Lazaric, A., Restelli, M., & Bonarini, A. (2008). Transfer of samples in batch reinforcement learning. In *Proceedings of the 25th international conference on machine learning* (pp. 544–551). ACM.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444. doi:10.1038/nature14539

LeCun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. E., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, *1*(4), 541–551. Retrieved from http://dblp.uni-trier.de/db/journals/neco/neco1.html#LeCunBDHHHJ89

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the ieee* (Vol. 86, *11*, pp. 2278–2324). Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.7665

LeCun, Y., Bottou, L., Orr, G. B., & Müller, K.-R. (2012). Efficient backprop. In G. Montavon, G. B. Orr, & K.-R. Müller (Eds.), *Neural networks: Tricks of the trade (2nd ed.)* (Vol. 7700, pp. 9–48). Lecture Notes in Computer Science. Springer. Retrieved from http://dblp.uni-trier.de/db/series/lncs/lncs7700.html#LeCunBOM12

Lee, H., Grosse, R. B., Ranganath, R., & Ng, A. Y. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In A. P. Danyluk, L. Bottou, & M. L. Littman (Eds.), *Icml* (Vol. 382, p. 77). ACM International Conference Proceeding Series. ACM. Retrieved from http://dblp.uni-trier.de/db/conf/icml/icml2009.html#LeeGRN09

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., … Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 1–14. doi:10.1561/2200000006

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml* (Vol. 30, *1*).

Mitchell, T. M. (1997). *Machine learning*. McGraw-Hill.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., … Kavukcuoglu, K. (2016). Asynchronous Methods for Deep Reinforcement Learning. *arXiv*, *48*, 1–28. Retrieved from http://arxiv.org/abs/1602.01783

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. Retrieved from http://arxiv.org/abs/1312.5602

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. a., Veness, J., Bellemare, M. G., … Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. doi:10.1038/nature14236

Nair, V. & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In J. Fürnkranz & T. Joachims (Eds.), *Icml* (pp. 807–814). Omnipress. Retrieved from http://dblp.uni-trier.de/db/conf/icml/icml2010.html#NairH10

Parisotto, E., Ba, L. J., & Salakhutdinov, R. (2015). Actor-Mimic: Deep Multitask and Transfer Reinforcement Learning. *CoRR*, *abs/1511.0*. Retrieved from http://arxiv.org/abs/1511.06342

Pavlovsky, V. (2017). Introduction to convolutional neural networks. Retrieved June 4, 2017, from https://www.vaetas.cz/blog/intro-convolutional-neural-networks/

Perkins, T. J., Precup, D. et al. (1999). Using options for knowledge transfer in reinforcement learning. *University of Massachusetts, Amherst, MA, USA, Tech. Rep.*

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 386–408.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. Retrieved from http://dx.doi.org/10.1038/323533a0

Rusu, A. A., Rabinowitz, N. C., Desjardins, G., Soyer, H., Kirkpatrick, J., Kavukcuoglu, K., ... Hadsell, R. (2016). Progressive Neural Networks. *arXiv*. arXiv: 1606.04671. Retrieved from http://arxiv.org/abs/1606.04671

Selfridge, O. G., Sutton, R. S., & Barto, A. G. (1985). Training and Tracking in Robotics. In A. K. Joshi (Ed.), *Ijcai* (pp. 670–672). Morgan Kaufmann. Retrieved from http://dblp.uni-trier.de/db/conf/ijcai/ijcai85.html#SelfridgeSB85

Smolensky, P. (1986). Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. In D. E. Rumelhart, J. L. McClelland, & C. PDP Research Group (Eds.), (Chap. Information Processing in Dynamical Systems: Foundations of Harmony Theory, pp. 194–281). Cambridge, MA, USA: MIT Press. Retrieved from http://dl.acm.org/citation.cfm?id=104279.104290

Spong, M. W. (1995). The swing up control problem for the acrobot. *IEEE control systems*, *15*(1), 49–55.

Sunmola, F. T. & Wyatt, J. L. (2006). Model transfer for markov decision tasks via parameter matching. In *Proceedings of the 25th workshop of the uk planning and scheduling special interest group (plansig 2006).*

Sutton, R. S. & Barto, A. G. (1998). *Reinforcement Learning: An Introduction.* Cambridge: MIT Press.

Sutton, R. S., Mcallester, D., Singh, S., & Mansour, Y. (1999). Policy Gradient Methods for Reinforcement Learning with Function Approximation. *In Advances in Neural Information Processing Systems 12*, 1057–1063. doi:10.1.1.37.9714

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., ... Rabinovich, A. (2014). Going deeper with convolutions. *CoRR*, *abs/1409.4842*. Retrieved from http://arxiv.org/abs/1409.4842

Tanaka, F. & Yamamura, M. (2003). Multitask reinforcement learning on the distribution of mdps. In *Cira* (pp. 1108–1113). IEEE. Retrieved from http://dblp.uni-trier.de/db/conf/cira/cira2003.html#TanakaY03

Taylor, M. E. & Stone, P. (2009). Transfer Learning for Reinforcement Learning Domains : A Survey. *Journal of Machine Learning Research*, *10*, 1633–1685. doi:10.1007/978-3-642-27645-3

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, *8*, 257.

Tibshirani, R. (1996). Regression shrinkage and selection via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 267–288.

Tieleman, T. & Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning.*

Walsh, T. J., Li, L., & Littman, M. L. (2006). Transferring state abstractions between mdps. In *Icml workshop on structural knowledge transfer for machine learning.*

Williams, R. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, *8*(3), 229–256.

Wilson, A., Fern, A., Ray, S., & Tadepalli, P. (2007). Multi-task reinforcement learning: A hierarchical bayesian approach. In Z. Ghahramani (Ed.), *Icml* (Vol. 227, pp. 1015–1022). ACM International Conference Proceeding Series. ACM. Retrieved from http://dblp.uni-trier.de/db/conf/icml/icml2007.html#WilsonFRT07

Xu, B., Wang, N., Chen, T., & Li, M. (2015). Empirical evaluation of rectified activations in convolutional network. *CoRR*, *abs/1505.00853*. Retrieved from http://dblp.uni-trier.de/db/journals/corr/corr1505.html#XuWCL15

Zou, H. & Hastie, T. (2003). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, *67*(2), 301–320.