Vrije Universiteit Brussel

Faculteit Wetenschappen en Bio-ingenieurswetenschappen
Departement Computerwetenschappen

Dealing with the Dimension of Time in Deep Reinforcement Learning

Proefschrift ingediend met het oog op het behalen van de graad
van Master in de Ingenieurswetenschappen: Computerwetenschappen

Ruben Vereecken

Promotor: Prof. Dr. Peter Vrancx
Prof. Dr. Ann Nowé

August 2016

Vrije Universiteit Brussel

Faculty of Science and Bio-Engineering Sciences
Department of Computer Science

Dealing with the Dimension of Time in Deep Reinforcement Learning

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

Ruben Vereecken

Promotor: Prof. Dr. Peter Vrancx
Prof. Dr. Ann Nowé

Augustus 2016

Abstract

Deep learning is an increasingly popular topic in the area of machine learning. Reinforcement learning has recently adopted it as a method for learning behavior from raw sensory input such as image frames. Good results have already been achieved but many problems still elude us, especially where dependencies over time must be learned in order to achieve results.

In this thesis, we adopt the Arcade Learning Environment to train agents that learn how to play Atari games based solely on raw pixel input and a reward signal. In order to be successful these agents need to learn important features autonomously, not only visually but also temporally.

The current state of the art, Deep Q-Network (DQN), only takes into account the most recent frames. We explore three alternative neural network architectures and their capabilities of handling dependencies over time and compare these against DQN.

Two of the networks considered here are valid alternatives for learning short-term, fixed-time features just like humans can intuitively derive the direction something is moving in. The first aims to encompass time information by learning from two input frames that correspond to different time steps and merging that information in a deeper layer.

The second involves convolutions over time and is shown to significantly outperform the standard approach on a problem with considerable amounts of obscured information.

The last architecture under consideration makes use of long short-term memory, which effectively equips the network with internal memory that can hold past information.

Acknowledgements

First and foremost I would like to thank Prof. Dr. Peter Vrancx for the countless times we have sat together, brainstorming for interesting problems and discussing how to approach them. His expertise always managed to guide me in the right direction and his patience never failed to baffle. This thesis would not have existed as it does without the inspiration offered as well as all the invaluable advice he was always willing to impart.
Secondly, I would also like to thank Prof. Dr. Ann Nowé for the opportunity.
Lastly, I would like to thank my mother for her endless support and bottomless faith in anything I have ever undertaken, not the least of which this dissertation.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CHAPTER

1

INTRODUCTION

Reinforcement Learning is what we have labeled the remarkable ability of man and beast alike to learn from their surroundings and their interactions with it. It is embodied in our learning from past mistakes or our failing to do so. For quite some time now people have attempted to bestow this same ability on our computing devices so that they would figure out how to tackle problems by themselves without explicit instructions.

Autonomous learning agents are already present in the periphery of our day to day activities. For example, networks can learn to route smartly depending on time of day and congestion or any other available information. More and more, autonomous learners are even invading the commercial and personal space. Smart homes can learn to anticipate the inhabitants' behavior and proactively control the thermostat, leading to a better experience and lessening the ecological footprint of a home without the intervention of humans.

Recent years have also seen a rise of interest in robotics and control, a popular example of which is the ongoing research concerning self-driving cars (Google, n.d.) along with the multitude of uses for an autonomous robot even if only in a restricted and controlled environment.

Clearly, in this day and age where progress is considered one of the most noble of goals, reinforcement learning could conceivably offer the tools for a great many things we might consider *of the future*.
The field is relatively young yet has already booked remarkable advances. Still there is a very real sense of unfinishedness. We do not hint at immaturity for it has already reached the point at which it can be considered mature but rather that the field has not reached its full potential. The applications that have been conceived successfully are not as widespread as they could be and those applications we aim to achieve still partially elude us. Of course, for a researcher this poses incredibly interesting opportunities.

The field of reinforcement learning inches forward and manages the occasional leap. In recent years, some of the greatest contributions make use of deep learning, a concept borrowed from machine learning, within the reinforcement learning context.
Deep Learning originates in supervised learning and concerns itself with learning end-to-end, hierarchical representations of data which can manage unprecedented performance and are especially interesting because they can eliminate the need for handcrafted features. A great overview of some of the most important contributions and noteworthy applications in this regard is given by Lecun, Bengio, and Hinton (2015).
The field has had successes in face recognition (Lawrence, Giles, Tsoi, & Back, 1997; Wolf, 2014), speech recognition (Dahl, Yu, Deng, & Acero, 2012; Graves, Mohamed, & Hinton, 2013) and genomics (Leung, Xiong, Lee, & Frey, 2014; Xiong et al., 2015) and is far from limited to these alone.

Recent successes in reinforcement learning include successfully transferring deep learning principles outside of their original context.
A great effort has recently been put forward by Mnih et al. (2013, 2015) with the introduction of the Deep Q-Network. Through the combination of great research and sufficient computation power their learning agents have managed to beat humans on an array of Atari games. Still, some games remain largely or completely unbeaten. The scientific community jumped on the opportunity to further improve on DQN and to explore the area that now goes by the name of deep reinforcement learning.

The current state of the art reinforcement learning can now learn straight from raw image pixels given a sufficiently deep neural network to represent its value function. While spatial features can be learned hierarchically,

the dimension of time has been mostly neglected or simplified. This is because much of the theoretical work is based on the requirement that all state information that could be useful is present; no hidden state allowed, not even historical state that is not observable anymore. A short-term example of this is velocity which can only be derived when multiple frames are present. A more complicated, long-term example would be the game Blackjack where the cards used in previous rounds affect the possibilities in later rounds.

Experiments with Atari games manage to circumvent the requirement somewhat since games are too simple to contain such long-term dependencies. All information one could need is usually encompassed within a few frames at any point, like the velocity of a moving object on the screen can be calculated from two frames.

This neglect and oversimplification of the dimension of time struck as interesting and as an area deserving of investigation. In this thesis, we set out to compare different approaches to handling the dimension of time in reinforcement learning.

1.1 Research Question

In this thesis I set out to investigate what the effect of time dependencies is in a reinforcement learning setting and how to successfully learn these in order to learn control policies. In particular, the problems I consider are all Atari games where the agent needs to decide on an action given solely raw pixels as input along with a game-specific reward signal to guide its actions. What makes these games interesting is that sometimes state is not available within a single frame; one does not know which way the laser is moving from just looking at a standstill.

The current standard approach to cope with features that can be derived over time is to feed multiple frames into the architecture at once. This already is an improvement over using just a single frame because the network suddenly has more information available. The idea behind passing in multiple frames is that the network will learn to combine the information available in multiple time steps. However, it should be obvious that this approach is rather naive and limited to only a small fixed history.

I will consider three alternative approaches. Late Fusion and 3D convolutional networks are of the same variety: they too employ a fixed window but do so differently. Both have successfully been applied in a machine learning context before (Karpathy et al., 2014; Ji, Yang, Yu, & Xu, 2013). The last and third approach involves long short-term memory, which has already been shown to be able to remember dependencies over arbitrary amounts of time (Bakker, 2001), albeit it in a vastly more simple setting.

My contribution consists of applying these approaches in a deep reinforcement learning setting and comparing them in different scenarios against one another and against the standard approach.

1.2 Outline

This thesis will gradually build up sufficient understanding for the reader to understand the contributions presented. Chapter 2 concerns itself with providing background concerning Machine Learning. Following with Artificial Neural networks is Chapter 3, which examines how to construct and train neural networks. Chapter 4 will provide a full understanding of the reinforcement learning setting, what is aimed to being achieved and how neural networks fit in. Chapter 5 follows naturally with more recent extensions to reinforcement learning and gives some necessary background to deep reinforcement learning.

The contributions of this thesis are mainly in Chapters 6 and 7. Chapter 6 outlines 3 architectures that are each in turn explored and discussed. At the end of Chapter 6 the reader can find an overview comparing these architectures with the standard approach and one another.
Chapter 7 will draw conclusions, outline my contributions and point to interesting avenues for future work.

Appendix A can be visited for experimental details and statistical tests while Appendix B contains additional experiments, not all of which are mentioned in the text.

CHAPTER

2

MACHINE LEARNING

2.1 Introduction

The world of today is generating huge amounts of data as it goes along, and more and more of it is put to use. It is quite probable that any consumer in our society that goes shopping, uses some kind of service, or uses the internet on a daily basis performs actions that end up in a database. Not only do persons generate data, so does everything else we monitor, and our society happens to monitor quite a lot. This data is invaluable for any organization that benefits from information, and truly, most do. It is used by financial institutions like insurance companies, medical institutions, commercial companies, scientific research and everything in between. Evidently, as the need for knowledge (or awareness thereof) increases, so must our abilities to uncover it, making machine learning an incredibly interesting topic in today's world.

People are good at creating hypotheses about what things mean, connecting the dots and predicting cause or outcome. Yet we can only cope with so much data at a time and without direction, the answers we seek may elude us. Enter the computing device, programmable to our needs. Computer programs often take the form of our expert knowledge put to detailed

writing but they can also cover the gaps where we don't have the knowledge and instead need to uncover it. This includes anything of valuable information to us, be it patterns in certain data or hypotheses for predictions. Where the domain expert describes such hypotheses using a computer program, the machine learning expert devises ways of uncovering the hypotheses. Evidently, the power of computers would be much greater if they would be made able to learn as we do.
Great advances have already been made in the domain, yet we have not managed to make computers learn as well as humans do, at least not in as many regards as we do. The best applications do, however, get better results than humans in very specific domains.

Machine learning is a collection of methods used to bridge the aforementioned gap between data and knowledge. A machine learning algorithm learns from data and learns an hypothesis or model on it, which can then be used to make predictions on future data. This separates it from other algorithms designed by an expert that follow a static rule set. It is inherently an interdisciplinary field, with its roots firmly embedded in a statistical foundation combined with AI, drawing inspiration from fields such as information theory, complexity theory, psychology and other fields.
The rest of this chapter will describe the basics of machine learning along with the specific tools I will use throughout this thesis. The foundation should be quite sufficient to build up an understanding of the field so as to allow the reader to follow the following chapters comfortably. Next to exploring a foundation, I will describe specific methods such as neural networks and convolutional layers, both exciting techniques that will be used further on. However, perhaps to the reader's delight, a cursory glance at the high-level introduction of these methods should suffice for the reader who only seeks to understand the applications described herein. The careful reader is of course cordially invited to read the more detailed descriptions.

2.2 Hands-on Introduction

A machine learning algorithm is usually required to form hypotheses based on given observations. In this way it can be seen as a black-box oracle in which data are shoved and answers pop out. Let's take as an easy example the probably overused and dead-beaten weather forecast.
Let's say we only need to do a daily forecast. Data come in in the form of

meteorological observations such as temperature, wind speed, and whatever else experts observe to make top-notch predictions. This then gets paired with a description of the actual weather the next day. Together they form experience-target pairs (x_i, y_i) that correspond to a meteorological observation on the one day and the weather on the next day, in the hope that we can somehow deduce the hidden link between the two. The target y_i is what we're really after so we should choose carefully as to what we are trying to predict. y_i can be a nominal value, i.e. from a given set of values such as the set $(sunny, rainy, windy)$, in which case the problem is called a *classification* problem. Contrarily, it can be a real-valued number such as temperature in which case we call the problem a *regression* problem. The distinction is worth noting because some techniques lend themselves well to one category but not necessarily to the other.

On this data we will try to learn a hypothesis h that can predict the target weather y for any given observation x. Now all we need to do is fill in the bits in between.
We already have a vague description of what we want to achieve but now we need to design the problem. Let us pick for our targets the real-valued temperature on a given day. The observations, or *features*, we will base our model on will take the form of a vector $(Temperature, Sky, Wind)$ or (x_1, x_2, x_3) short, where $Temperature$ and $Wind$ are real-valued, whereas Sky takes on one of the three values $(Sunny, Rainy, Cloudy)$. Now we need some kind of function connecting observation to target. One of the easiest choices would be a linear function that combines the features and outputs the target we are looking for. It would look quite simple:

$$y = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3$$

We are still stuck with the problem that x_2, or Sky, is a nominal value and therefore does not fit all too well with algebra. We can work around this by agreeing on a little preprocessing step that maps $Sunny$ to -1, $Rainy$ to 0 and $Cloudy$ to 1. Now that our model is defined, all that is left for us to do is train it.

In order to train our model we need a data set with training examples (x_i, y_i), where the temperature y_i corresponds to the observations x_i the previous day. We now need to pick the best weights so the resulting model is the best hypothesis. We describe a *best fit* we need some metric of performance. The easiest one available is the error on the predictions made by

the model compared to the targets in the training data. The best hypothesis is the one that minimizes this error. The most common error by far is the *mean squared error*, though others certainly exist. Given a training set D, it is defined as follows:

$$E = \frac{1}{|D|} \sum_{(x_i, y_i) \in D} (h(x_i) - y_i)^2$$

Multiple algorithms exist that minimize E in such a manner. If all data is available in one go, as it is in our case, we can simply apply linear regression to our system of $|D|$ equations $y = xw$ in order to find a w with the lowest error.

2.2.1 Finishing up the example

An alternative algorithm to find our optimal weights w is the *weight update rule*, a source of inspiration for many more algorithms. The core idea is that every sample you calculate your prediction and shift the weights in the direction of the error, proportionally to the size of the error. So, for every sample:

1. Calculate $\hat{y_i} = h(x_i)$
2. Update $w_i = w_i + \eta(y_i - \hat{y_i})x_i$

The update size can be tweaked with η.

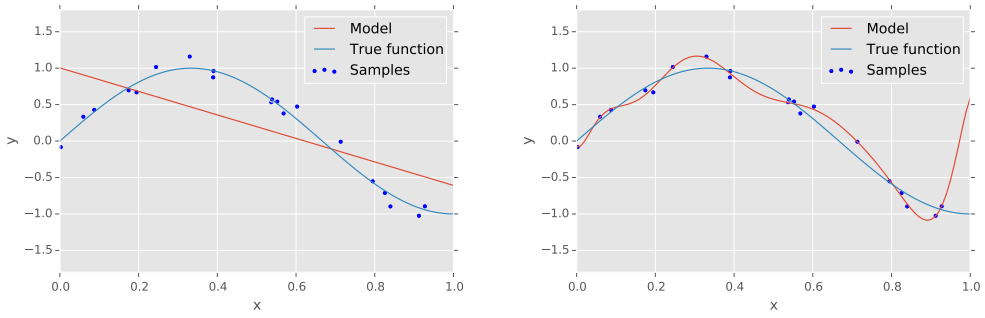
A theoretical basis for this is given in section 3.4.

As you have witnessed yourself, designing a machine learning task is rather an art than an exact science. The first optimization algorithm proposed, linear regression, is only available for linear models. Finding the ultimate weights makes more sense in a regression context as opposed to a classification context.
There are many more such decisions to made and models to choose from. During the next sections, I will touch on some interesting models that are used throughout this thesis.

2.3 Underfitting and Overfitting

Why not just go for an error of 0? We could, however it is unlikely since we suffer from two problems: insufficient data and a delusion of a perfect world. In other words, our features probably are not sufficient and our data is noisy. If we still manage a training error of 0 we probably committed the grave mistake of *overfitting* the training data, meaning we included the noise of the data in our model. The reason this is to be avoided is that by definition noise cannot be predicted and a model that tries to do so will perform even worse on similar unseen samples.

Figure 2.1b demonstrates this phenomenon of overfitting. A high-dimensional polynomial is fitted on a small set of samples drawn from a noisy distribution that follows a goniometric function. The model that we would like to attain should be as close as possible to the original function. However, if we try to fit the available data completely, the polynomial results in a model that performs well in the very close vicinity of the training points but performs horrendously on unseen data that deviates from the original training data. This can be seen from the discrepancy between the model and true function. It follows that overfitting is characterized by a low training error but a high generalization error, or put otherwise, the model fails to generalize the training data.



(a) Underfitting (b) Overfitting

Figure 2.1: Underfitting and overfitting demonstrated with two polynomial models on noisy data modeled after a goniometric function.

The opposite of overfitting would then be *underfitting*, as demonstrated by Figure 2.1a. This is often the case for models that lack representational power no matter how much data is available. While the figure contains a

rather naive example of a 1-dimensional polynomial fit, it illustrates perfectly how some models can never hope to attain a complex underlying function.
It will not be possible to minimize the training error entirely, though an underfitting model may still outperform an overfitted one.

As should be clear from the example, neither under- nor overfitting are good, in fact an algorithm designer should be weary of both. It is best to explore multiple algorithm settings in order to find a model that is prone to neither kind of misfitting.
In the next section I will describe a measure that helps guide this search.

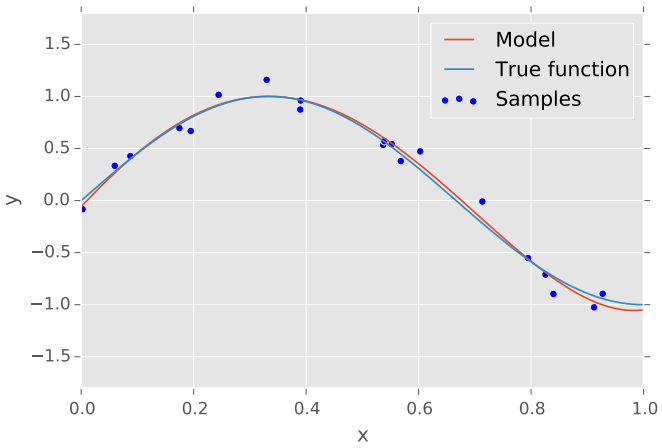


Figure 2.2: The right model can perfectly generalize from noisy data, insofar as the data covers the desired domain

2.4 Bias and Variance

A common way to describe a learning algorithm is in terms of its bias and variance. They are two different sources of error the algorithm designer wishes to minimize. The *bias* of a learning algorithm is the error inherent to the algorithm because of false assumptions, such as an unsuitable degree for a polynomial model in the previous example. This does not necessarily mean the model is weak on its own, it only is for a specific problem. Bias gives cause to underfitting, as can be seen in Figure 2.1a.

Bias can be quantified as the difference between the expected prediction and the true function value:

$$\text{Bias}(x) = E[\hat{f}(x)] - f(x)$$

Variance on the other hand, is error because of sensitivity to noise: small fluctuations in the data set. It is the variability of a model's predictions. As a result, high variance may causes overfitting noise inherent to the data but not to the model. A typical example can be seen in Figure 2.1b.
Variance can be quantified in terms of the expected deviation of a prediction (variability):

$$\text{Variance}(x) = E[(\hat{f}(x) - E[\hat{f}(x)])^2]$$

It can be lowered by reducing the amount of features (especially those of little import to the model) or increasing the amount of data points.

Bringing these two notions together gives us the *bias-variance decomposition* of a learning algorithm's generalization error, i.e. the expected error on unseen data:

$$\text{Err}(x) = E[(Y - \hat{f}(x))^2]$$

$$\text{Err}(x) = (E[\hat{f}(x)] - f(x))^2 + E[(\hat{f}(x) - E[\hat{f}(x)])^2] + \sigma^2$$

As you can see, the first two terms are Bias squared and Variance respectively. That leaves us with the last term which is the irreducible error, an error resulting from a lack of omniscience; a lack of data.
From this decomposition we can see that we can see that we can only have an error of 0 given the original model behind the data an infinite amount of data.

Figure 2.3 visualizes the bias-variance decomposition and shows the general tendencies of its individual components. As bias decreases, variance generally goes up and vice versa. The goal of any algorithm designer is then to find the sweet spot; the model that optimizes the traded between the two.
This can be done by exploring parameters for a certain model, like the degree of a polynomial, and calculating the *expected generalization error*, since the true generalization error is never known. This in turn can be done by keeping a separate test set and calculating the error on that, or

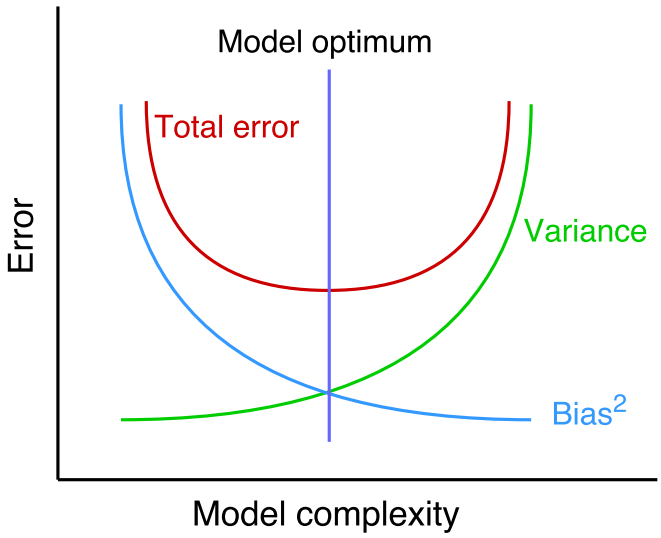


Figure 2.3: Bias-Variance decomposition

other techniques that make clever use of the training data such as bootstrap or cross-fold validation. I'll delve into those in the next section.

So far the discussion on the generalization error and how to mitigate it, I'd like to finish with a depiction of bias and variance in Figure 2.4. The center of the red target is some target value $f(x)$ and darts are predictions $\hat{f}(x)$. You can imagine this is the same x for many instantiations of the same model or it is a different x for every dart $\hat{f}(x)$, it doesn't matter.
A high variance will have a scattering effect. After all, variance describes the variability of predictions. High bias is the inability to generalize well because of the model's false assumption: in the image the model assumes an incorrect target. The goal is once to have both measures as low as possible.

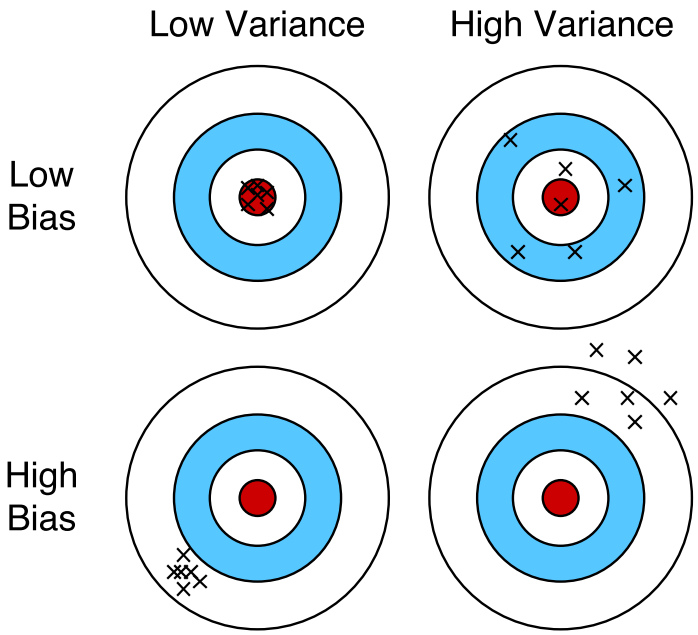


Figure 2.4: Dartboard Analogy (Sammut & Webb, 2011)

2.5 Model Validation

When training a model, one typically wants to know the generalization error as it is described in the previous section. You already have the training error because it is what you are trying to minimize, but it usually generalizes poorly to the generalization error. It can actually be harmful to watch only the training error during training, so you need an indication of how the model generalizes past training.
Often there are training, validation and test phases. Validation is the stage where we are trying to decide on the best model; this is when model parameters are determined (model selection). During testing we simply see how well the model is expected to generalize beyond experience. At this point no more tuning is performed. If the same data would be used for both validation and testing, the resulting test error would probably be overly optimistic (Friedman, Hastie, & Tibshirani, 2001) since we are trying to determine prediction error based on the data that was used to determine the very model we are using.

A naive way to go about this is to take a separate set of data that comes from the same distribution as the training data (because otherwise your measure

says very little or your training data was chosen poorly) and calculate the error of the model on this set. This *validation set* should be large enough to be representative of the underlying distribution but need not be as large as the original training set. A common measure, when dividing a data set, is to keep 20% to 30% aside for validation purposes. If a test set is also required, a good rule of thumb is keeping 50% for training data and dividing the other 50% equally over validation and test sets.
This way of splitting up data is far from ideal because it requires that the validation data not be used during training. Data is however far too valuable for training, we would rather not use it to simply compute a measure of success.

2.5.1 Cross-Validation

Cross-validation or K-fold cross-validation is by far the most popular validation technique. Gone with the separate validation set (I'll ignore testing for prediction error for now), cross-validation testing solves the problem of data scarcity by not using up the data you desperately need for training.

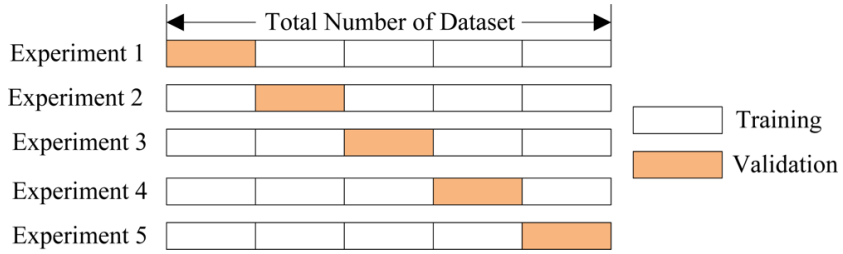


Figure 2.5: 5-fold cross-validation (Zhang & Wu, 2011)

Figure 2.5 illustrates how the training set is now divided into K equal-sized *folds*. Each of these folds will be a validation set in its own time.
The technique requires K different models to be learned, for a total of K iterations. During iteration i, fold i is kept aside as the validation set for this iteration and all other folds together comprise round i's training set.
For each model the error is calculated on the validation set, let's call this CV_i. The cross-validation error then becomes:

$$CV(\hat{f}) = \frac{1}{N}\sum_{i=1}^{N} CV_i$$

We now have a guess at the prediction error. A good K is paramount for a good cross-validation error, it also has a huge impact on the required amount of work since computation time grows linearly with K. We would like it to be quite low to save on computation time yet high enough to have the cross-validation error be a meaningful estimate. Lower K tend to higher bias because less data is used for training. Higher K then tend towards high variance. This comes down to the same trade-off between bias and variance, though this time of an estimator that estimates generalization error. Generally good compromises for K are 5 and 10 (Kohavi et al., 1995).

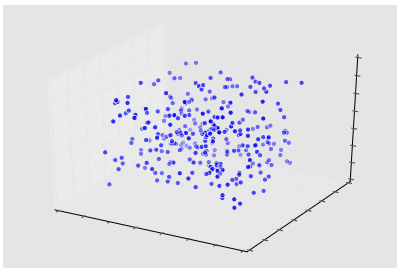
2.6 Curse of Dimensionality

The distance between two instances is calculated based on all of their attributes. For these instances to have a small distance between them, they should be close to one another in all of their attributes, or otherwise put, they should be similar in all aspects. The more attributes or dimensions they have, the harder this becomes. This informally described phenomenon we call the *curse of dimensionality*.

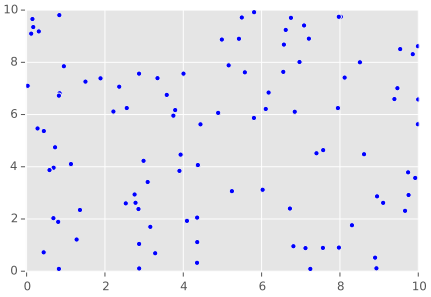
Figure 2.6 attempts a visual demonstration. Each plot contains 100 instances with a variable amount of dimensions, but with values for all dimensions ranging uniformly between 0 and 10. As instances gain more dimensions, the spaces containing them become increasingly more sparse. Even though an instance may be close to another instance in some dimensions, any discrepancies in other dimensions will only increase the distance between the two instances.

Let us resume with the above example. Another way to uncover the increasing sparsity is to examine is by examining the density of each space. The 1-dimensional space has 10 instances per unit of space, since it contains 100 instances spread over 10 units. In contrast then, the 2-dimensional space contains only a single instance per square unit of space and the 3-dimensional space a forlorn .1 instance per cubic unit of space. In order to boost the density of the 3-dimensional space to be on par with the 1-dimensional one, we would need a hundred times as many instances in this particular scenario, for that is the factor between the two space sizes.

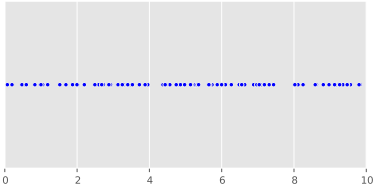
A final measure of sparsity of a space I like to think in is the expected average distance for dimensions, given their distributions. In other words, if one were to pick two instances from one of the above spaces uniformly, what would the expected distance between the two be? This measure re-



(a) 3 dimensions



(b) 2 dimensions



(c) 1 dimension

Figure 2.6: Each plot contains 100 points with values in all dimensions ranging uniformly between 0 and 10. The curse of dimensionality is especially clear from the difference between the 2-dimensional and 1-dimensional plot.

lates directly to the curse of dimensionality; as the number of dimensions increase, expect the distances to go up as well.

The phenomenon is often encountered in areas such as text processing or image processing where large numbers of dimensions are inevitable. A naive model that tries to represent texts could have a different dimension for each word, likewise a model that describes an image would have a separate dimension for each pixel, easily reaching into thousands of attributes. Such models would need a tremendous amount of instances to have any sort of meaningful distance measure. Acquiring the required quantities of data is often infeasible, making techniques that shrink space dimensionality very valuable in practice. Techniques can range from discarding least important attributes to statistically combining attributes to even learning new attributes that represent higher-level concepts from the lower-level attributes. A popular example of the latter are convolutional networks, a topic I will cover in section 5.1.

CHAPTER

3

ARTIFICIAL NEURAL NETWORKS

Neural networks provide a robust method to learn vector-valued target functions. They have effectively been used for a multitude of machine learning domains, including recognition of handwritten characters (Y. LeCun et al., 1989) and face recognition (Cottreil, 1991), but also in reinforcement learning (Anderson, 1989; L.-J. Lin, 1993). Both popularity and effectiveness of neural networks can be attributed to their ability to cope with noisy, real-world data, as well as being able to approximate any function, albeit with possibly very large complexity (Mitchell, 1997, chapter 4).

In part, they are man's attempt to model the biology of the brain, or at least that is where the inspiration is drawn from. The artificial neural networks we construct consist of interconnected units, each taking in real-valued inputs and outputting a real value, possibly connecting to multiple other units. This can be in a directed, feed-forward manner, but it does not have to be. Interesting architectures have been discovered that make use of recurrent connections. Such architectures will be considered in section 5.3. To complete the analogy, the human brain consists of a large amount of densely packed neurons, approximately 10^{11} in total. Compared to an artificial neuron, they fire a lot slower yet still humans are capable of making

incredibly complicated decisions (or recognitions) in a matter of less than a second. This hints at great parallelism, an attribute that is still important in the artificial neural networks we construct, especially in real-life situation reinforcement learning where decisions must be made swiftly and we cannot wait for a slower model to finish computing. While our general purpose machines are entirely sequential in nature, parallel hardware has been built specially for neural network applications. These are not generally available, as each domain and even each problem still require their own specific architectures. Still, in modern day, many make use of the parallel capabilities of Graphic Processing Units (GPUs), effectively making artificial neural networks even more useful.
The analogy is limited however. Our use of neural networks draws inspiration from nature but does not aim to mimic perfectly. Whereas our neural networks output real values, biological neurons produce a series of spikes where both timing and intensity impact the represented information (Bialek, Rieke, De Ruyter Van Steveninck, & Warland, 1991). The reader interested in spiking neuron networks is referred to a good overview by Paugam-Moisy and Bohte (2012). For now, we shall content ourselves without the dimension of time in this regard.

3.1 Building Blocks

Artificial neural networks are made up of simple elements called units. Units are connected through directed connections that are associated with a weight. These weights are what make up the parameters of the model. Units are usually divided in layers depending on their function and location within the network. Non-input units get their activation signal, i.e. their input, from some function of the values of the incoming connections and the weights associated with them. We call these functions *activation functions*. They are usually non-linear and play an important role in how the input value is expressed, or *activated*.

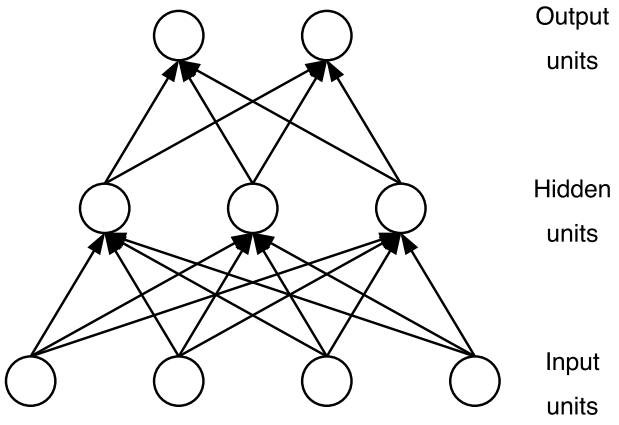


Figure 3.1: Multi-layer neural network with a single hidden layer.

3.2 Activation Functions

Activation functions are used to transform the output of a unit in a neural network. They have a crucial impact on the shape of the output space, so some care should be taken when deciding on one.
Formally, an activation function ϕ will compute

$$o = \phi(\overrightarrow{w} \cdot \overrightarrow{x}) \tag{3.1}$$

where $\overrightarrow{x}$ is the vector of incoming values from other units and $\overrightarrow{w}$ is the vector of associated weights.

3.2.1 Perceptron

Activation functions can be linear or nonlinear. A special kind of unit is the *perceptron* (Rosenblatt, 1958), which outputs either a -1 or 1, depending on the value of a linear combination of the input:

$$o(\overrightarrow{x}) = \begin{cases} 1 & if \overrightarrow{w} \cdot \overrightarrow{x} > 0 \\ -1 & otherwise \end{cases} \tag{3.2}$$

where $\overrightarrow{w}$ denotes the weights of the incoming connection and $\overrightarrow{x}$ the corresponding incoming values. This function effectively creates a hyperplane decision surface; it separates the input space in two.

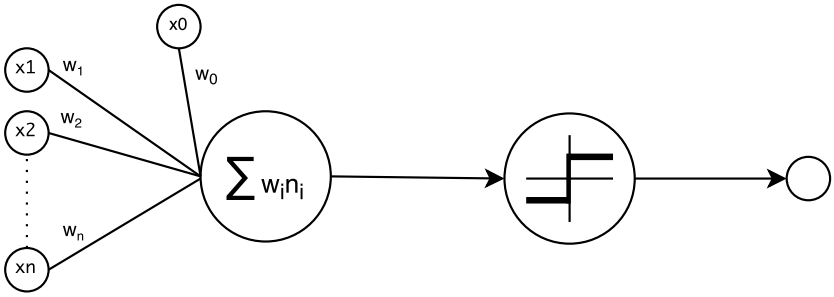


Figure 3.2: Basic perceptron unit.

3.2.2 Sigmoid

A problem with perceptrons is that they are not differentiable in their whole domain. As will become clear in section 3.4, differentiability is a useful and ofttimes required attribute. As a way to circumvent this difficulty of non-differentiability, the *sigmoid* (denoted $\sigma(x)$) is often used:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.3}$$

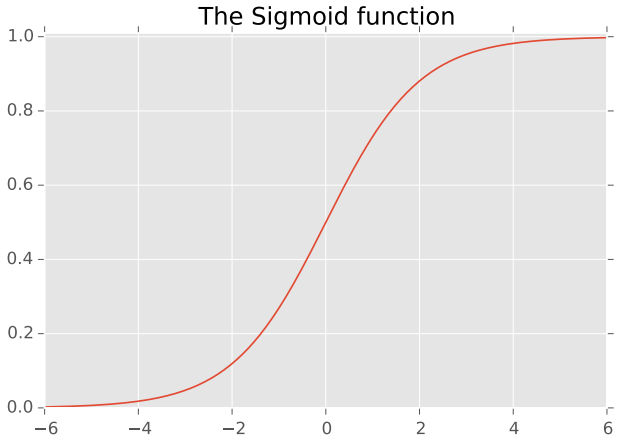


Figure 3.3: The Sigmoid function

As can be seen on fig 3.3, it has the same tendency as the perceptron activation function to separate the input space in two but there is a transition area of uncertainty between the two extremes, allowing the function to be differentiable in its whole domain.

In practice, the range is normalized between -1 and 1 just like the perceptron.

3.2.3 Hyperbolic Tangent

As a cousin to the sigmoid function, the hyperbolic tangent deserves mentioning as well. It bears the same shape as the sigmoid function since it is really only a stretched and shifted version.

$$htan(x) = \frac{1}{1 + e^{-2x}} + 1 \tag{3.4}$$

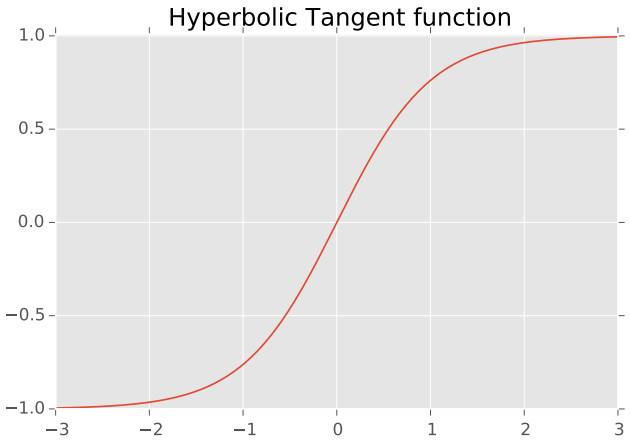


Figure 3.4: The hyperbolic tangent function

3.3 Rectified Linear Unit

A fairly recent activation function is the Rectified Linear Unit, or ReLU for short (Nair & Hinton, 2010).

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases} \tag{3.5}$$

While it is very simple in its essence, it has some important properties that can be leveraged. Among the advantages is that anything below zero is mapped to zero. This is great for sparse network activations. In a scenario with weights initialized symmetrically (e.g. uniformly) around 0,

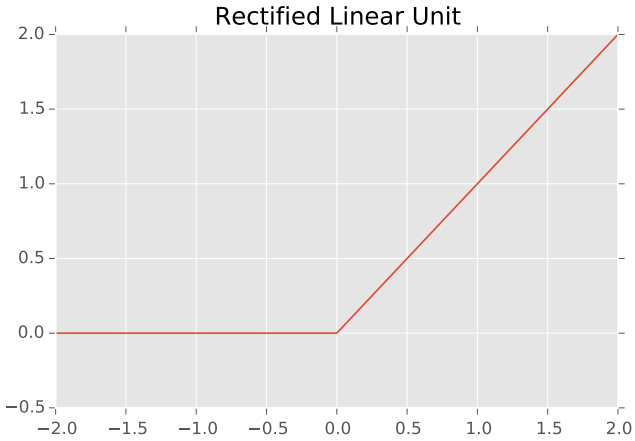


Figure 3.5: Rectified Linear Unit

only half the neurons will activate. This sparsity is great for discrimination and effectively makes training neural networks faster, making it especially interesting for deep learning (Lecun et al., 2015).
Linear rectifier functions also help mitigate the *vanishing gradient* problem since gradients above zero are simply propagated (**Glorot2011**). A more detailed discussion on the vanishing gradient problem can be found in section 5.3 pertaining recurrent neural networks.

The rectified unit family has gained quite some attention ever since the first promising results. Leaky rectified units comprise a subfamily of functions that are designed to have a non-zero gradient over their entire domain. The basic version simply has a fixed slope below zero that is kept constant throughout training.
One such variant is the *Parametric rectified linear unit* (He, Zhang, Ren, & Sun, 2015), a unit with a parametric slope below zero that can be trained on data.
Another variant is the *Randomized rectified linear unit* which has a randomized slope below zero. The supposition is that the noise will help prevent overfitting during training, so the slopes are fixed outside of training.

3.4 Gradient Descent and Backpropagation

Earlier we talked about a learning algorithm's error and even glossed over ways to minimize this error. In this section we will discuss gradient descent as a general approach followed by backpropagation, a specific gradient descent algorithm designed to minimize errors for artificial neural networks.

The approaches discussed here make certain assumptions about what the algorithm's error looks like in function of the parameters of the algorithm. In other words, they make assumptions about the error surface.
Differentiability is even a requirement, meaning the derivative of the error exists in each point, so for each combination of algorithm parameters. This gives rise to smooth surfaces (though possibly steep) that do not contain gaps, such as the one in Figure 3.6.
Despite the requirement, the methods described here still perform badly on irregular surfaces: surfaces where the gradient changes sign often for example, so surfaces with many local minima.

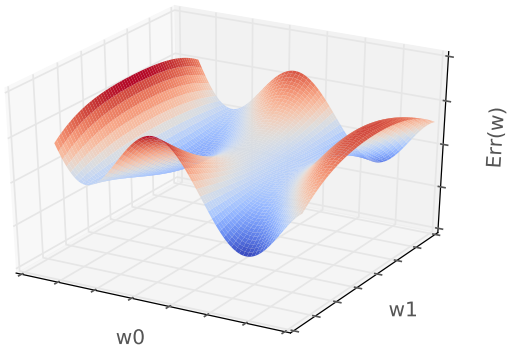


Figure 3.6: Error surface in function of two parameters of the model that are to be learned.

3.4.1 Gradient Descent

Gradient descent or steepest descent can be described as the general approach of moving across an error surface by following the path with the greatest gradient - the path with the steepest slope - downward. This makes

it a greedy algorithm. Following such a slope means adjusting the parameters (or weights) in the direction of the slope.

As stated before, gradient descent requires the derivative of the error E to exist in the whole domain. Formally, in function of the weight vector $\overrightarrow{w}$ of length n, we write the derivative or gradient of E as:

$$\nabla E(\overrightarrow{w}) = \left(\frac{\delta E}{\delta w_0}, \frac{\delta E}{\delta w_1}, \ldots, \frac{\delta E}{\delta w_n}, \right) \tag{3.6}$$

$\nabla E(\overrightarrow{w})$ is in itself a vector and corresponds to the steepest slope I described earlier, with both a size and direction. Since this gradient describes the steepest *increase* of E with respect to $\overrightarrow{w}$, we need to use the negated gradient $-\nabla E(\overrightarrow{w})$ to follow the slope downward.
We can now use this for our weight update:

$$\Delta \overrightarrow{w} \leftarrow \overrightarrow{w} - \eta \nabla E(\overrightarrow{w}) \tag{3.7}$$

We call η the learning rate, useful for tuning the step size of the algorithm. Smaller values will converge more slowly while larger ones have the danger of overstepping local minima, making convergence even slower. Gradient descent is actually known to 'zigzag' near convex valleys. Because of this, the learning rate is sometimes reduced over time so the learner is less prone to overstep and instead settle into minima.

In order to actually compute the gradient iteratively, we still need a way to calculate the partial derivations $\frac{\delta E}{\delta w_i}$. We still have not defined the error E, so let us start with that. For a training data set D and learning algorithm $\hat{f}$, we define the error E w.r.t. to $\overrightarrow{w}$ as:

$$E(\overrightarrow{w}) \equiv \frac{1}{2} \sum_{(x_i, y_i) \in D} (y_i - \hat{f}(x_i; \overrightarrow{w}))^2 \tag{3.8}$$

The reason why it is chosen so specifically will become clear in a moment.

In order to algebraically derive E w.r.t to w, we also need to define the learning algorithm f, which of course plays a key role in what the error surface looks like. It is possible to approximate the gradient near a certain point x in case the algebraic derivation is too hard to calculate or the calculations are too complex. However, we will content ourselves with a simpler example that allows us to perform an algebraic derivation and calculate the gradient without such methods.

For f, we will pick a simple linear unit that will suffice as an example:

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

Finally then, the differentation of E:

$$\begin{aligned} \frac{\delta E}{\delta w_i} &= \frac{\delta}{\delta w_i} \frac{1}{2} \sum_{(x_j, y_j) \in D} (y_j - \hat{y}_j)^2 \\ &= \sum_{(x_j, y_j) \in D} (y_j - \hat{y}_j) \frac{\delta}{\delta w_i} (y_j - \overrightarrow{w} \cdot \overrightarrow{x_j}) \\ &= \sum_{(x_j, y_j) \in D} (y_j - \hat{y}_j)(-x_{ij}) \end{aligned} \tag{3.9}$$

with x_{ij} the i'th component of x_j. As you can see, the derivation is comfortably easy because of our choice of the linear unit and the way the error function is defined. Other errors but most of all other learning algorithms can have far less obvious differentiations.
Combining this with 3.7 gives us the update rule for a specific weight component:

$$\Delta \overrightarrow{w}_i = \eta \sum_{(x_j, y_j \in D)} (y_j - \hat{y}_j) x_{ij} \tag{3.10}$$

Equation 3.10 describes the gradient descent algorithm for the linear unit. Each iteration the weight update gets calculated for all the training examples by computing $\hat{y}_j$ and following 3.10, then summing all weight updates w_{ij} to get w_i. Only once all weight deltas Δw_i are calculated should the actual weight vector be updated (with Δw comprised of the individual w_i of course):

$$\overrightarrow{w} = \overrightarrow{w} + \overrightarrow{\Delta w} \tag{3.11}$$

This then gets repeated until some termination condition is met, like a sufficiently small error or a desired amount of passes over the training set.

Because first the whole weight update is calculated with respect to the whole training data set and only then the weight vector is adjusted, this vanilla version of gradient descent is also called *batch gradient descent*. It can be quite inefficient and downright impractical for large training sets that do not fit into memory. For this reason, the stochastic variant is often used.

3.4.2 Stochastic Gradient Descent

A variation on regular gradient descent that alleviate the problems with regular stochastic gradient descent's convergence speed and even tendency to fall into local minimum pitfalls (Mitchell, 1997, chapter 4) is *stochastic gradient descent*.
Instead of performing the parameter updates in batch fashion, parameters are updated incrementally. This means that for every training sample (x_j, y_j), $\overrightarrow{w}$ gets updated immediately and therefore affects the calculation of the next $\Delta\overrightarrow{w}$. Stochastic gradient descent basically descends towards the steepest gradient of the error with respect to a single example. This makes it much faster than the regular batch version and allows it to be used for online learning. It also means that stochastic gradient descent does not use the true gradient to adjust parameters, instead it uses some complicated estimate.
It should be obvious that it is computationally cheaper because it performs a step for every sample instead of once for the whole sample set, but it can also sometimes avoid falling into local minima because it uses a different gradient than regular gradient descent does.

By adjusting η to be smaller as compensation, stochastic gradient descent can be made to approximate the true gradient arbitrarily closely. Even better, it has been shown that slowly decreasing the learning rate, as proposed before, will allow stochastic gradient descent to match the convergence behavior of its batch counterpart.

3.4.3 Backpropagation

We have now laid out the basis for the *backpropagation* algorithm, a learning algorithm designed for multilayer networks that are capable of representing even the most nonlinear surfaces. Backpropagation (Rumelhart, Hinton, & Williams, 1988) will allow us to learn the weights of a multilayer network by minimizing a measure of difference between network output and target output. This minimization will be done with gradient descent and the measure of error we will employ is

$$E(\overrightarrow{w}) \equiv \frac{1}{2} \sum_{(x_j,y_j)\in D} \sum_{k\in outputs} (y_{kj} - \hat{y}_{kj})^2 \tag{3.12}$$

where y_{kj} is the *k*th element of the output value that corresponds to input sample x_j, and $\hat{y}_j$ is the network value for x_j,
so $\hat{y}_{kj}$ corresponds to its kth value. Having the error defined like this allows gradient descent to minimize the error across all training samples.

The network should have an activation function as described in Section 3.2; we will refrain from picking a specific example for now and call this activation function ϕ.
However, since backpropagation is based on gradient descent, it has the same requirement of differentiability of the function in question. Since the network is made up partly of these activation functions, we require these be differentiable. Because of this, backpropagation can not readily be used with a unit such as the perceptron because it is not differentiable.

In the algorithm, we will need a delta that signifies how off the network's output is. The δ_n described here are actually the negative derivatives of our error w.r.t. to the network in node n, but we will not go into a full explanation of how this result was achieved. I only mention this because this $-\frac{\delta E}{\delta net_n}$ corresponds closely to the negative gradient we used in gradient descent.

First off, create a neural network and initialize all network weights. Initialization can be done in different ways, according to different distributions, but a popular standard way of doing things is to initialize them all according to a small uniform distribution centered around zero.
Once created, feed the given training sample through the network, giving o_k as output for every unit k. The units o_k in the outer layer together make up $\hat{y}$.
Computing the error term δ_k for an output unit k and a given training sample is pretty straightforward:

$$\delta_k = \psi(o_k)(y_k - \hat{y}_k) \tag{3.13}$$

with $(y_k - \hat{y}_k)$ simply the difference between the target output and the network output and ψ the derivative of the activation function ϕ. For example, if we were to use the sigmoid function σ (Section 3.2.2):

$$\psi(x) = \sigma(x)(1 - \sigma(x)) \tag{3.14}$$

and since o_k is calculated by the activation function σ:

$$\psi(o_k) = o_k(1 - o_k) \tag{3.15}$$

The δ_k in 3.13 describes the error for an output unit, but how do we compute the error for a hidden unit, since a hidden unit has no provided target value?
It turns out it should depend on the error terms of the units it connects to, weighted by the weights associated to those connections. For a hidden unit h, write its error term δ_h as:

$$\delta_h = \psi(o_h) \sum_{k \in outputs} w_{kh}\delta_k \tag{3.16}$$

where w_{kh} denotes the weight associated to the connection from unit h to unit k. These are the weights that of course have to be tuned, so intuitively backpropagation will tune the responsibility or impact of each node for/on the next node, effectively learning hidden representations about the hypothesis in question.

Now that we have defined the error terms for all weights, we can calculate the weight update for the weight associated with the connection from unit i to unit j:

$$\Delta w_{ji} = \eta \delta_j x_{ji} \tag{3.17}$$

Just as with gradient descent, this can be done until some termination criterion is reached, be it convergence or amount of iterations.
In the stochastic gradient descent setting of this algorithm, Δw_{ji} would be used immediately to update w_{ji}. This is how backpropagation is commonly encountered but one could also choose to employ the batch version just as was the case with regular gradient descent. In that scenario, sum all Δw_{ji}s gotten for all examples of the training set and only then perform weight updates.
In practice this is never done because the approach scales horribly with training set size. In fact, backpropagation is usually unleashed on the training set multiple times in order to converge to a local minimum, seeing every sample more than once.

Momentum

A common extension to backpropagation is momentum. Every weight update, you carry over some of the weight update from last iteration:

$$\Delta w_{ji}(t+1) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(t) \tag{3.18}$$

We call α in 3.18 the *momentum*. It is aptly named after the physical property and describes how a moving object needs a larger force to be applied (or for longer) in order for it to come to a standstill, the larger the momentum (and analogously for getting up to speed, though we conveniently ignore that).
As a result, a high momentum will keep the weight vector rolling in the same direction in flat regions and even roll up small slopes thanks to momentum accumulated previously, great for escaping local minima overcoming flat plateaus.
As an extra, convergence is sped up because regions with unchanging downward slopes (negative gradients) will accumulate a lot of speed.

3.4.4 Adagrad and RMSProp

Some parameters get updated less frequently than others while they may play a key role for the others, they are the proverbial needs in the haystack of parameters. Adagrad (Duchi, Hazan, & Singer, 2011), its name derived from 'adaptive gradient', addresses exactly this need of allowing important yet less updated parameters to play a bigger role in gradient updates. It builds on stochastic gradient descent but adds the notion of parameter-specific learning rates that are continuously updated to reflect the scarcity of updates for a parameter.
Let us start by introducing a shorthand for the gradient of the error at time step t for a parameter vector θ:

$$g_t \equiv \nabla_\theta E(\theta) \tag{3.19}$$

Our stochastic gradient descent update rule then becomes:

$$\theta_{t+1} = \theta_t - \eta \cdot g_t \tag{3.20}$$

Now we introduce a diagonal matrix G_t, which I will discuss right after:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{3.21}$$

Note that $\odot$ denotes element-wise matrix-vector multiplication and ϵ is simply a tiny non-zero value to avoid division by zero.

The matrix G_t is a diagonal matrix of which each element i, i corresponds to the sum of squares of the gradients corresponding to θ_i up to time step t.
The element $G_{t,ii)}$ looks as follows.

$$G_{t,ii} = \sum_{\tau=1}^{t} g_{\tau,i} g_{\tau,i}^{\top} \tag{3.22}$$

RMSProp

Adagrad effectively takes into account the past updates to a parameter. However, since it uses all past gradients the values of G_t do tend to blow up over long series of samples and as a result the learning rate vanishes almost entirely, preventing the algorithm from learning anything beyond that point. Multiple approaches have been developed to tackle this problem.
One such is *RMSProp*, introduced by Hinton, Srivastava, and Swersky in a lecture in his Coursera Class (2012). It solves the ever-growing G_t and the resulting ever-diminishing learning rate by keeping a moving average of the squared gradient for all parameters:

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2 \tag{3.23}$$

The update rule then becomes

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{3.24}$$

This is what it is named after, as it keeps the Root Mean Squared error in the denominator and propagates it by keeping a moving average.

CHAPTER

4

REINFORCEMENT LEARNING

> Fool me once, shame on you;
> fool me twice, shame on me.
>
> Popular proverb

4.1 The Problem

Reinforcement Learning in general is the idea of learning from interaction with the environment; a concept humans are familiar even if sometimes only subconsciously.
A child learns to walk by attempts, failure, and eventually success. During every interaction with our environment we are constantly aware of how it reacts to us, be it when we walk down the street or hold a conversation with someone. We are even aware of animals doing the same.

The idea of trial-and-error learning has long been in play, the term itself even used in the 19th century to describe observations of animal behavior (Woodworth & Schlosberg, 1938).

Edward Thorndike phrased the *Law of Effect* which is, after some of his own amendments, still considered a basic principle governing much of animal behavior.

> Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the ani- mal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond.
>
> (Thorndike, 1911, p. 244)

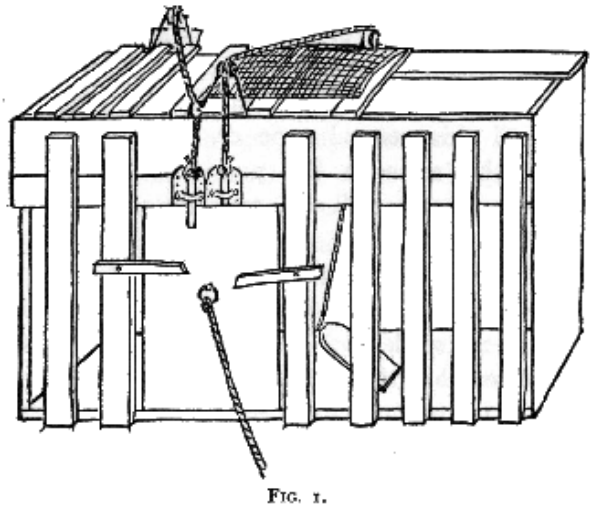


Figure 4.1: A puzzle box used by Thorndike in his experiments. The hungry cat is locked inside the box which it can open by solving some kind of puzzle in order to reach the fish outside. Results showed that the cat went from solving the puzzle by sheer happenstance to methodically opening it as if by habit.

The psychological field of animal behavior and learning has a longer history than the computational counterpart does, with popular examples such as Pavlov's research. The Russian Nobel laureate studied how animals responded to different stimuli:

> It is pretty evident that under natural conditions the normal animal must respond not only to stimuli which themselves bring immediate benefit or harm, but also to other physical or chemical agencies—waves of sound, light, and the like—which in themselves only signal the approach of these stimuli; though it is not the sight and sound of the beast of prey which is in itself harmful to the smaller animal, but its teeth and claws.
>
> (Pavlov, 1927, p. 14)

In this same text the term "reinforcement" was used for the first time in the context of animal learning.
Pavlov is most known for his experiment with dogs. He noticed dogs would produce saliva in response to receiving food. He then associated a secondary stimulus to the act of feeding the animals by ringing a bell beforehand. The dogs learned to associate the bell with food and produced saliva upon hearing the bell, even before or without receiving anything.

Using trial-and-error to achieve artificial intelligence was among the earliest ideas in the field. Alan Turing describes positive and negative stimuli to influence an algorithm:

> When a configuration is reached for which the action is undetermined, a random choice for the missing data is made and the appropriate entry is made in the description, tentatively, and is applied. When a pain stimulus occurs all tentative entries are canceled, and when a pleasure stimulus occurs they are all made permanent.
>
> (Turing, 1948)

Reinforcement Learning as we treat it here solely means the computational approach to learning from interaction, except where mentioned explicitly. We will not theorize on animal behavior or try to model it in order to create computational models. Sometimes, however, inspiration is drawn from animal behavior but it usually is no more than that; analogies only go as far as they serve a goal. The perspective used here is that of the engineer, not the neuroscientist.

Central to reinforcement learning is that a learner interacts with its environment in order to learn and ultimately aims to achieve some goal. Applied to human behavior in the course of a lifetime, we could say humans

-in general- try to optimize happiness. Applied to something less daunting than the human condition, a sculptor may try to optimize beauty or expressivity of a sculpture, learning along the way how to do so in the best possible way. It is this goal-based interaction that forms the core to reinforcement learning.

Reinforcement learning can be characterized by three distinguishing features, setting it apart from other fields of learning

Closed loop In order to learn, a learning agent must interact with the environment to collect the necessary information. However, each action changes the environment in a certain way and in turn influences the agent's future inputs. This forms a *closed loop*.

Discovery An agent is provided with no instructions on what actions to take and how this will impact the environment. It is to discover this itself.

Time factor Consequences of an action can be delayed by an unknown amount of time. Even the reward signal can be received many time steps later. The agent is to figure out for itself how its actions relate temporally to consequences.

A good example of this is the game of chess with only a reward signal at the end of a match, either positive or negative. Some moves during the game are probably more important than others and some more complicated, like traps that take multiple moves to set up. Yet all moves together are rewarded with only a single signal at the end of the match. It is for the agent to unravel and attribute its actions according to importance.

4.1.1 Reinforcement Learning and Machine Learning

As I tried to convey above, a crucial aspect to reinforcement learning is trial-and-error, learning from interactions with an environment that is not necessarily known. This makes reinforcement learning different from *supervised learning* which is most associated with machine learning. In a supervised setting, a labeled example set is provided for the learner to learn from. Learning in this context means generalizing from the training set so queries about data not in this set can still be answered accurately.

Reinforcement learning is different in that it takes on more of the problem; a learning algorithm is not presented with data but instead must gather it by interacting with the environment. In doing so it must also make a trade-off between exploration and exploitation, a characterizing element to reinforcement learning. A learning agent can either *exploit* what it knows to be the best action in order to achieve its goal, thereby possibly ignoring alternative routes of action that would have resulted in better results, or choose to *explore* what impact its actions have on the environment. Neither strategy can be followed exclusively if one wants to learn anything worthwhile, a good combination of both is always needed.

On the "opposite" side of supervised learning is what we call unsupervised learning, which is about finding hidden patterns in unlabeled data. The supervision gap between the two obviously pertains to whether the data has been created in a supervised manner. An illusion is created that there are two sides to machine learning, though reinforcement learning does not seem to fit either. Reinforcement learning is effectively a third machine learning paradigm.

4.2 Reinforcement Learning Framework

4.2.1 Agent and Environment

There are only few components to the reinforcement learning problem. The learner and actor is dubbed the agent and everything outside it the environment. The agent perceives its environment and acts on it, receiving a reward in return. The latter is a numerical value which the agent tries to gain as much as possible of during its time interacting with the environment. This interaction goes on continually: observation followed by action followed by reward. The goal of the agent is to maximize its accumulated reward over the entire span of a task, i.e. an instance of a problem. In order for it to do so it must learn to not only look to immediate rewards but must also look to what the future has to offer.

Formally, we divide the problem into discrete time steps t. A time step occurs every time the agent perceives a state S_t from the set of possible states $\mathcal{S}$. Based on this state the agent selects an action A_t from its repertoire of actions $\mathcal{A}(S_t)$ available to it at in state S_t. As a result, the next step it will

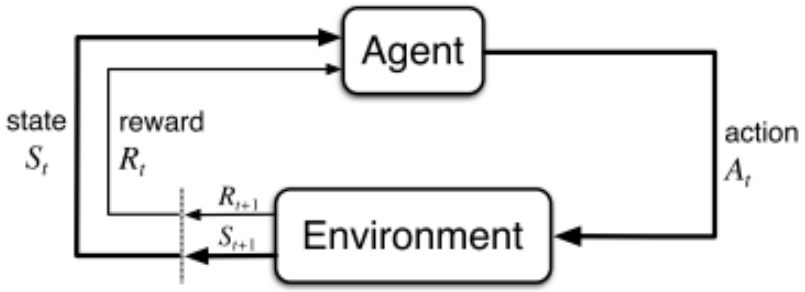


Figure 4.2: The agent interacts with the environment and consequentially perceives a reward and the next state of the environment. (Sutton & Barto, 1998)

receive a numerical reward $R_{t+1} \in \mathcal{R}$ along with the next state S_{t+1}, and so on and so on.

As you can see, the problem setting is entirely abstract and can be filled in in various ways. In fact, the same practical problem may be defined in different ways depending on the goal. A state could just as well represent a robot's raw sensor readings as well as higher-level representations such as whether it is looking at a green or red light. We say that state is provided by the environment even though one could argue a robot's sensors are part of the agent. Instead, we look at the agent as the observer and decision-maker taking in everything from a distance, even its physical body external to it as part of the environment. Similarly to states, actions can range from raw low-level values like motor voltages to higher-level concepts like which room to go to or whether to turn left or right. Basically, actions are decisions taken by the distant observer, for the designer to decide which shape they take.

Cart Pole Balancing ExampleA popular experiment in reinforcement learning that has its roots close to half a century ago is the cart pole balancing experiment (Michie & Chambers, 1968).
In this experiment, a pole is affixed to a joint to the cart. Because of the joint, the pole can move in a two-dimensional plane as shown in Figure 4.3. The cart can move in two directions parallel to the joint's movement.
Let us say the pole starts in a perfectly orthogonal position to the cart. However, because of the joint and inherent material imperfections, however microscopic, the pole will start to drop toward one side or the other. It

is now the cart's goal to keep the pole balanced perfectly upward by moving however it can.

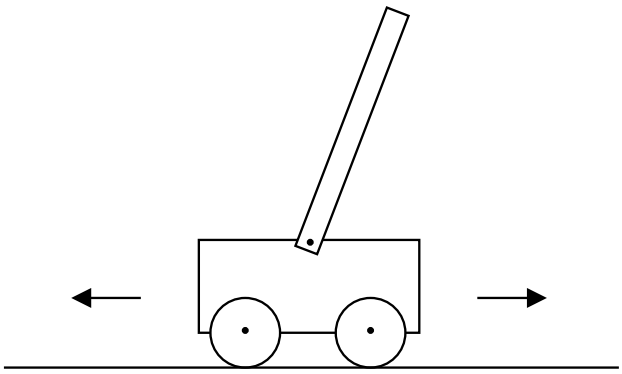


Figure 4.3: Cart Pole Experiment

Applied to our framework, we could at first glance say the cart is the obvious agent and everything else its environment. It would however make more sense to name the textitdecision maker that controls the cart the agent and anything physical the environment or observable state.
As it is commonly defined, a state exists of the angle of the pole with the cart along with the angular velocity of the pole. This would mandate three possible actions: move left, move right, or stand still.
However, say the cart works in a more complicated manner and actually has to accelerate instead of just move at a fixed speed in one direction. In that case, the aforementioned state is insufficient to correctly balance the pole; car speed is now a crucial component and should be added to the state if possible to give a more complete description and allow the agent to learn even better policies. If not added, the agent will still learn to the best of its abilities though.

4.2.2 Goal and Rewards

During a task, an agent receives rewards upon acting with the environment. Rewards are crucial to the reinforcement learning problem as they are way the algorithm designer can create goals for the agent (the *what*) without stating how these goals might be achieved (the *how*). In the earlier example of a chess game, you would reward the agent for winning (and even penalize for losing) but in general not reward specific steps the agent took to reach its goal of winning. That would be bringing domain specific

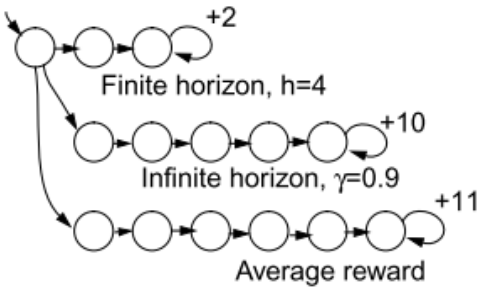


Figure 4.4: Different goal models (Kaelbling, Littman, Moore, & Hall, 1996)

knowledge into it, which is not necessary and can even be detrimental to the learner's success. The most basic form contains no domain knowledge.

Rewards are numerical and can be positive or negative, real-valued or restricted to natural numbers, the agent just has one goal with them: to somehow accumulate it. There are different ways to define such a goal.

finite horizon The most naive way to collect rewards is through the *finite-horizon* model, in which the agent aims to optimize its expected rewards for the next fixed amount of steps. This is rarely appropriate and still leaves us with the challenge of finding an appropriate value for the constant horizon window.

infinite horizon The alternative then is to not have a fixed window but try to maximize reward over the rest of the agent's lifespan, albeit discounted so later rewards have less impact than current ones.

average reward The average reward model makes the agent optimize its average reward over the rest of its lifetime. It relates closely to the infinite-horizon model, without the discount.

We will only use the *infinite-horizon* model. Over the course of a lifetime the agent will try to optimize
the expected total discounted return

$$E\left(\sum_{t=0}^{\infty}\gamma^t r_t\right) \tag{4.1}$$

or put otherwise, at time step t the agent will try to optimize his total discounted reward G_t by choosing the appropriate action A_t:

$$G_t = \sum_{k=1}^{\infty} \gamma^k R_{t+k} \tag{4.2}$$

As stated, we will discount later rewards to avoid infinity as time goes on. This discounting is tuned by $\gamma \in [0, 1]$, the *discount rate*. Though a nifty mathematical trick, it also has the interpretation of giving more weight to rewards closer in the future than those further down the road. In case of $\gamma = 1$, an reward later on has the same impact on the goal as an equally high reward encountered earlier. Tuning γ effectively tunes the agent's farsightedness, with lower values for γ make for a 'greedier' agent.

Cart Pole Balancing Example (cont.) Let us take again the cart pole balancing example. We still have to split up the problem in discrete time steps. Take for example 40 steps every second, because that is the limitation imposed by the hardware and it does not make sense to make decisions faster than they can be acted upon.

We now need to convey to the cart agent exactly what we want it to achieve: to keep the pole balanced upward indefinitely. A good way to get the goal across is to define a range within which the robot is doing a good job, then always assign $+1$ whenever the pole is inside that range and -1 when the pole drops too low. This will make sure the agent tries to get within the positive-reward range but potentially also as far as possible from the negative-reward fringes, depending on how the agent learns. If the positive range is then defined around the center, as it should be, the pole should be balanced perfectly upward.

4.3 Markov Decision Processes

The Markov property is an interesting property for a problem to have because it states that all relevant information at some point is present in a state, a state being whatever is available to the agent. This means that the whole history leading up to a certain point, insofar that it is relevant, is encoded in the state at that time. Such a state is said to be Markov.

Consider the probability of ending up in a state S_{t+1} with reward R_{t+1} after performing some action A_t in a state S_t. This probability is denoted

$$Pr(S_{t+1} = s', R_{t+1} = r | S_t, A_t) \tag{4.3}$$

We would say a state signal is Markov if this probability is the same as

$$Pr(S_{t+1} = s', R_{t+1} = r|S_t, A_t, S_{t-1}, A_{t-1}, ..., S_0, A_0) \tag{4.4}$$

which is the same scenario except given the entire state and action history. If these two are indeed equal then the history is effectively encoded in the current state S_t.

Markov states are handy because we do not need the whole history in order to make good decisions or even the best decisions. The best policy in function of the entire history is the same one as without the history because everything relevant to predicting the environment is already encoded in the current state. This is especially important for reinforcement learning as decisions will always be made in function of only the current state. Even when the state is not entirely Markov, it is easier for us to think of it as Markov or an approximation thereof because it gives us a framework on which we can build an understanding, so long as we are conscious of possible deviations.

4.3.1 Markov Decision Processes

Once a reinforcement learning task has the Markov property we call it a *Markov Decision Process* or MDP for short. Such an MDP is completely defined by its probability of transitioning to any state s' with reward r when starting in any state s and acting with some action a:

$$p(s', r|s, a) = Pr\{S_{t+1} = s', R_{t+1}|S_t = s, A_t = a\} \tag{4.5}$$

And if one would like to know the chance of transitioning to a state regardless of the reward associated to the transition (the *state-transition probability*) it could be easily calculated as:

$$\begin{aligned} p(s'|s, a) &= PrS_{t+1} = s'|S_t = s, A_t = a \\ &= \sum_{r \in \mathcal{R}} p(s', r|s, a) \end{aligned} \tag{4.6}$$

Another especially useful number is the expected reward for a certain state-action pair:

$$
\begin{aligned}
r(s,a) &= E[R_{t+1}|S_t = s, A_t = a] \\
&= \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} p(s', r|s, a)
\end{aligned} \tag{4.7}
$$

As should be obvious from these examples, the Markov assumption is an extremely convenient one.

4.3.2 Partially Observable Markov Decision Processes

One approach to non-Markovian settings are *Partially Observable MDPs* (POMDP). They can be seen as generalizations of Markov decision processes with incomplete information (Lovejoy, 1991).
Formally, we can say there is an underlying MDP to the POMDP with states $\mathcal{S} = \{s1, s2, \ldots, s_N\}$ and a transition probability as described above. The agent can however not observe a state directly, at time step t it instead observes an observation o_t that relates to s_t in some way, according to some unknown probability distribution $P(o|s_t)$.

One way that POMDPs arise often is because of noisy sensors, so observations do not completely accurately reflect true state. Another example is when state features can be derived from observable features. A typical example of such features are derived features such as speed of an object as it relates to the object's position. Speed is often a crucial component but it is not always part of a state and indeed one could argue it should not be because it is derived from another part of the state, over time.

Many approaches to POMDPs include building some kind of model of the environment so the relation between observation and hidden state can be used (Cassandra, Kaelbling, & Littman, 1994). Other approaches use some kind of history window, both fixed (McCallum, 1995) and variable-width windows (Ring, 1994) have been attempted.
In section 5.3 we will explore another road with recurrent neural networks.

Cart Pole Balancing Example (cont.) In the cart pole example, we originally said we could have both angle and angular velocity of the pole as part of the state. What of the scenario when we do not supply angular velocity to the agent, instead it must rely only on the current angle?
The agent does not know how fast the pole swings, not even in which direction. All it knows is the current angle. Ideally it would know the

direction of the pole so it could counteract or complete its movement, but in order to do that it must now look at the history of the pole: which direction does it come from? What is the distance traveled since last step? This is clearly not Markovian, as in the markovian case all necessary knowledge is encoded in the current state, whereas this scenario requires knowledge of at least the previous step. Indeed, if the previous state could be part of the current state the agent would already have a wealth of information. We have:

$$\begin{aligned} & Pr\{S_{t+1} = s', R_{t+1} | S_t = s, A_t = a\} \\ \neq\ & Pr\{S_{t+1} = s', R_{t+1} | S_t = s, A_t = a, S_{t-1} = s'', A_{t-1} = a''\} \end{aligned} \tag{4.8}$$

This fixed-window approach is exactly what has been attempted successfully by L. Lin and Mitchell (1992), though they state it only works in cases where the window is sufficient to encompass the required history.

4.4 Value Functions

The *policy* of an agent is the decision maker, the component of the algorithm that decides on the action in each state. For a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}(s)$, $\pi(a|s)$ is the probability of taken action a in state s as directed by the policy π.

We now introduce the notion of a *value function*; a function that estimates how good a certain state is for the agent. Recall that the agent will try to optimize discounted accumulated reward G_t, this in turn depends on the future actions of the agent. Since actions are a result of the policy π, we always define a value function in terms of a policy:

$$\begin{aligned} v_\pi(s) &\equiv E_\pi[G_t | S_t = s] \\ &= E_\pi \left[\sum_{k=1}^{\infty} \gamma^k R_{t+k} \middle| S_t = s \right] \end{aligned} \tag{4.9}$$

We call this the *state-value function* for policy π.

Then there is the *state-action function* for policy π, denoted $q_\pi(s, a)$, which is the expected return when taking action a in state s then following policy π:

$$
\begin{aligned}
q_\pi(s,a) &\equiv E_\pi[G_t|S_t=s,A_t=a] \\
&= E_\pi\left[\sum_{k=1}^{\infty}\gamma^k R_{t+k}\middle|S_t=s,A_t=a\right]
\end{aligned}
\tag{4.10}
$$

Note that I still have not said how to calculate these functions. As they are defined here, they are the mathematical entities we aim to achieve and give us a way of thinking in ideal terms that we can then later approach in practice.

There is an extremely important recursive relation between value functions that forms the foundation for the sections to come.

$$
\begin{aligned}
v_\pi(s) &= E_\pi\left[\sum_{k=1}^{\infty}\gamma^k R_{t+k}\middle|S_t=s\right] \\
&= \sum_{\substack{s'\in\mathcal{S}\\ a\in\mathcal{A}(s)}} \pi(a|s)p(s'|s,a)\left[r+\gamma v_\pi(s')\right]
\end{aligned}
\tag{4.11}
$$

We call this relation the *Bellman equation*. Broken down, it is the sum of expected returns for each state s' accessible from state s, weighted by the chance of ending up in state s' (which is of course impacted by both the odds of picking action a and ending up in state s' after).

Using value functions, we can define a partial ordering between policies in order to decide on the best one. One policy is better than the other if its expected return is greater than the other's. Formally,

$$
\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s)\,\forall s \in \mathcal{S}
$$

We now have a front of *optimal policies*, which we will denote by $\pi*$. They all share the *optimal state-value function* and *optimal action-value function*:

$$
v_*(s) \equiv \max_\pi v_\pi(s) \qquad \forall s \in \mathcal{S} \tag{4.12}
$$

$$
q_*(s,a) \equiv \max_\pi q_\pi(s,a) \qquad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s) \tag{4.13}
$$

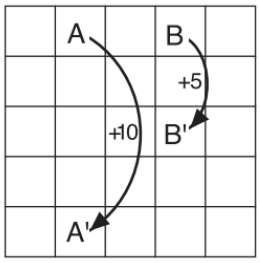
Finally, we can write the optimal action-value function in function of the optimal state-value function.

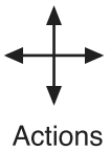
$$q_*(s,a) = E[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a] \tag{4.14}$$

It follows that an optimal policy is one that is greedy with regard to the optimal value function, i.e. one that always takes the action with which the highest value $q_*(s,a)$ is associated.

In practice it is rarely feasible to compute all required values in order to gain access to the optimal policy. State spaces may be too large or even infinite and the same may go for action spaces. The notion of optimality introduced here serves only as a theoretical framework on which we can build techniques to approach this optimality in varying degrees of success, heavily dependent on the problem statement at hand. In the next sections I will build up to such techniques that I will then use throughout this thesis.

Gridworld Example Another classic example in Reinforcement Learning is Gridworld. Figure 4.5a depicts the state space as a rectangular grid where each cell corresponds to a state. Only four actions are available to the agent in any state; it can move in each of four directions to a neighboring cell. There are three kinds of rewards. If the agent performs an action that would have it drop off the grid, it receives a penalty of -1 and remains in the same state. Moving to either state A or B gives the agent a bonus as depicted and moving it to a different state: A' or B' respectively. All other moves net the agent a reward of 0.





3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

Figure 4.5: Gridworld. Left are depicted the only two positive rewards. Right is v_π for the random policy. (Sutton & Barto, 1998)

Let us suppose now the random policy π which takes a random action at random in any and every state. Figure 4.5b then shows the value function v_π for this policy in the case that $\gamma = 0.9$. These values can be calculated

by solving the equation 4.13. For a small state space such as the Grid-world one, doing so explicitly is still possible. However, for larger ones this quickly becomes infeasible and one must turn to approximation methods which we will consider later on.
Note that the largest state values are at A and B. Still, relative to the reward associated with each, A's state value is not quite as high. This is due to the fact that A' is right next to the border of the grid and thus has more potential for falling off, i.e. for negative rewards which negatively impact the state values for states that can lead there.

4.5 Temporal Difference Learning

One of the core ideas to reinforcement learning is the concept of *temporal-difference* (TD) learning. TD methods do not require a model of the environment in order to work, they allow learning from raw experience which makes them all the more useful.

4.5.1 TD Prediction

We start out with the prediction problem: estimating the value function v_π for a policy π. After some experiences following the policy, the current estimates of v_π should be updated. All TD methods have in common that they update after each time step. The most simple method called $TD(0)$ updates its estimate V of the value function v_π for some policy at hand as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{4.15}$$

where S_{t+1} is the state transitioned to from S_t, while receiving a reward R_{t+1}. The factor α is a learning rate which impacts the size of updates to the value function estimate.
We call the TD method a *bootstrapping* estimate because updates rely on existing estimates, i.e. $V(S_{t+1})$ in 4.15. TD is also referred to as a *sample backup* because in order to create a new estimate for a state at time t, it uses an estimate of the successor state at time $t+1$ along with the reward achieved to get there. Every time a new state is reached, the value for the previous state is updated, hence the backup. In contrast, a *full backup* would require a full distribution of all possible successor states instead of just a single sample successor.

The obvious advantage to the TD method is that it requires absolutely no model of the environment, the reward signal or the distribution that governs next states when choosing an action. Instead, all knowledge is captured in the value estimates.
Another advantage is that TD methods are entirely online and experience an update after each and every step. This makes them applicable to any problem, be it an episodic one with very long episodes or a continuous one that never terminates. In contrast, other methods exist that update estimates after each episode. Care must be taken to adapt those methods to these problem statements but TD methods fit out of the box.
Finally it is worth noting that the estimates in the $TD(0)$ algorithm just described converge to the true v_π for any policy π, given that the policy remains fixed.

4.5.2 Q-Learning

Now that we have concerned ourselves with the problem of value prediction it is time to move to the *control problem*: that of finding an optimal policy. We must make a trade-off between exploration and exploitation, the characterizing dilemma of reinforcement learning that requires us to balance between exploiting what is known to be good or exploring the unknown in order to uncover even better. Two approaches exist, distinguished by the way exploration is handled: on-policy and off-policy. We will start with the latter in this section as this is the approach taken throughout this thesis, though I will touch on on-policy learning in section 4.5.3 to contrast the off-policy approach described here.

First off, we will concern ourselves with the action-value function instead of the state-value function. This because the control problem is about choosing actions, so we want to know the quality of each action at a given state. That is, we will estimate $q_\pi(s, a)$ instead of estimating $v_\pi(s)$ explicitly.

The algorithm of interest in this section, *Q-learning* (Christopher John Cornish Hellaby Watkins, 1989), is still one of the most important algorithms in reinforcement learning ever since its inception almost three decades ago. It is made especially interesting because it is proven to converge to the optimal state-action value function q_* given the requirement that all state-action pairs are visited and updated sufficiently along with a restriction on the step-size (Christopher J C H Watkins & Dayan, 1992).

At the core of Q-learning is its state-action value estimate update:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \quad (4.16)$$

We call this off-policy learning because the action-value function q being learned is not the one associated with the policy used to learn q. In fact, Q approximates q_* directly so the optimal policy being learned is the greedy one, independent of the policy used while learning.
This independence is achieved by the term $max_a Q(S_{t+1}, a)$ which maximizes for the greedy policy as it only considers the best known action to take in state S_{t+1}. In other words, it associated the quality of a state with the best action-value in that state.

The policy used during training is still important as it governs which state-action pairs will be visited. A greedy policy that only ever considers a few unique paths through the state-action space is probably a bad idea because it will neglect to update Q-values for other state-action pairs and as such will never converge to q_*.
One popular approach to balance the need for exploration versus exploitation is the ϵ-*greedy* policy. When faced with a choice between actions, it will select the best known action with a chance of $1-\epsilon$ and select a random action otherwise. The value for ϵ is usually kept low, like 0.1.
Often there is a need to explore a lot early, then exploit the gathered experience later on in order to find q_*. A typical example of this would be a game where a positive reward is only received when the game is won and a negative penalty is the result of early loss. The only positive reward might be hard to achieve without a long series of 'good' moves and even one mistake might cause an early loss, again preventing the agent from seeing a positive reward signal. Scenarios like this benefit from decreasing ϵ over time, for example in a linear fashion, until it reaches some smaller but still non-zero value that would cause the policy to exploit more as it gathers experience.

The full Q-learning algorithm with ϵ-greedy for a learning policy is displayed in Algorithm 1. Note that as it is described here it is a *tabular* algorithm, meaning Q is represented using a lookup table indexed by both a state s and action a. In next section we will proceed beyond the tabular case and investigate how to cope with state and action spaces too large for lookup tables.

Algorithm 1 Q-Learning

Initialize $Q(s, a) \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ (i.e. to 0)
for all episode **do**
  get start state S
  while S not terminal **do**
    if chance ϵ **then**
      $A =$ uniformChoice$(\mathcal{A}(S))$
    else
      $A =$ argmax$_a Q(S, a)$
    end if
    Perform action A, receive reward R and observe state S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \text{max}_a Q(S', a) - Q(S, A)\right]$
    $S \leftarrow S'$
  end while
end for

4.5.3 Sarsa

Opposite of off-policy learning we have on-policy learning. A prominent algorithm in this category is Sarsa (Rummery & Niranjan, 1994), so named because it uses the the following 5 events: $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$ as opposed to Q-learning, which does not take into account the *next* action A_{t+1}. The Sarsa update rule is as follows.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right] \tag{4.17}$$

When compared to Q-learning, it differs only in that it uses a different Q-value in the update. Instead of an ideal Q-value associated with the highest action-value for next state S_{t+1}, the Sarsa update rule also takes into account the action taken in that next state, i.e. A_{t+1}. This means one can only update an estimate Q after two actions performed, whereas Q-learning required only a single state-action-state transition.
The fact that Sarsa updates Q-values with respect to the learning policy makes it on-policy; it estimates q_π for the policy π at hand, not just the greedy policy, like Q-learning would.

4.6 Function Approximation

So far we have only dealt with the tabular case where state or state-action values could be stored in a table. However, as soon as we are faced with large state or action spaces it becomes intractable to fill the entire table, let alone iterate over its content sufficiently for it to approach the intended functions in a meaningful manner. We are faced with the problem of *generalization*, that of generalizing beyond learning experience to new situations while still maintaining useful value estimates.
Since we intend to generalize value functions, the generalization discussed here is called *function approximation*. What would have previously been table entries are now the samples of the function we aim to approximate. The techniques visited in this section are not unique to reinforcement learning. Generalization methods form the core of machine learning, so we can borrow extensively from that domain. The main question that still remains for us is how to employ these techniques for our reinforcement learning setting.

We start out again by estimating the state value function v_π associated with a policy π. This time however we do not use a table but instead some parameterized function $\hat{v}$ with parameters $\theta \in \mathbb{R}^n$. Our goal is to have $\hat{v}$ approximate v for all states $s \in \mathcal{S}$, we write $\hat{v}(s, \theta) \approx v_\pi(s)$. Similarly, we can aim to approximate the state-action value function $q_\pi(s, a)$ as $\hat{q}(s, a, \theta)$.

We can use any functional approximator introduced previously for v, like a simple polynomial where θ consists of the coefficients or even a neural network with θ all of the network's weights. The advantage of using such an approximator is that whenever a state is backed up, other states will have their values affected too because a single parameter usually affects the output for multiple or even all inputs. The idea behind this is that these changes beyond the single state result in generalization. Usually similar states are similar in quality so we will look for a function approximator that generalizes beyond a single state to similar states in a similar manner.

Not all traditional function approximation methods work just as well in the reinforcement learning setting. First off, data is gathered incrementally and each sample has a very strong correlation with the ones preceding and surrounding it. Not all approximation methods cope well with these strong correlations between data, they tend to introduce overfitting.

More importantly, in the machine learning setting, we would have a set of labeled samples that included input, approximation output but also a *target* output. The target output in this case would be the associated value function, v_pi in case of the approximator $\hat{v}$. This allows us to define an error, we will use the simple mean square error (MSE):

$$MSE(\theta) = \sum_{s \in \mathcal{S}} (v_\pi(s) - \hat{v}(s, \theta)) \tag{4.18}$$

Of course the error can be similarly defined where not v_π but q_π is to be approximated. While this gives us a theoretical setting to reason with, we do not actually have access to v_π and as such can not use it to generate target values. Instead we *bootstrap*, like before, using approximated values for each backup. This naturally brings along another problem. Not only are the targets bootstrapped, the targets also move as the learning process go on, because the targets themselves are affected each backup. This calls for approximation methods that are very robust to these non-stationary targets.
Let us return then to our error definition. As before, in the machine learning context, we want to minimize this. That is, we want to find a *global optimum*, a parameter vector $\theta*$ such that $MSE(\theta*) \leq MSE(\theta)$ for any possible θ. In order to minimize this we can reuse any of previously introduced methods, including for example backpropagation if our approximation method is a neural network.

An update for Q-learning after taking an action A in state S, then transitioning to state S' and receiving reward R, could look as follows.

$$\delta \leftarrow R + \gamma \max_a \hat{q}(S', a, \theta) - \hat{q}(S, A, \theta)$$
$$\theta \leftarrow \theta + \nabla \hat{q}(S, A, \theta)$$

This is of course gradient descent as described in section 3.4.1 with $\nabla f(\theta)$ again defined as the vector of partial derivatives as in equation 3.6.

4.6.1 Linear Methods

A simple approach that nevertheless can yield good results for problem domains that are not too complex is the linear model. In this case, θ is a

vector of n elements, corresponding to the feature vector $\phi(s)$ of length n. The approximation would then be:

$$\hat{v}(s, \theta) \equiv \theta^T \phi(s) = \sum_{i=1}^{n} \theta_i \phi_i(s) \tag{4.19}$$

Thanks to this definition the gradient of the value function approximation can be easily written as:

$$\nabla \hat{v}(s, \theta) = \phi(s) \tag{4.20}$$

This simple way of calculating the gradient for any given state makes linear methods extremely easy to use with gradient descent. Their popularity is further increased by having only a single optimum $\theta*$ (or multiple equally good optima) as opposed to multiple local optima. Any linear method that would get to local optima is thus guaranteed to converge to or near the global optimum.
As opposed to for example neural networks, linear methods are both quick to train and evaluate. Still, their domain of effectiveness is limited to approximately linear functions, so their power is severely limited when dealing with larger problems with complex non-linear relations between features (like say image classification based on raw pixels).

CHAPTER

5

DEEP REINFORCEMENT LEARNING

Deep Learning describes a family of learning techniques that at their core are about learning representations of data in a hierarchical fashion. It replaces the need for handcrafted feature and indeed works in an entirely unsupervised or at least semi-supervised fashion. In this way, it is often deployed as end-to-end learning because it can learn its own higher-level abstractions from raw data.
The core algorithm in deep learning is backpropagation (described in detail in 3.4) which describes how a learning model should update its parameters in response to a discrepancy between learned output and example output.

While the notion of stacking layers of units to create multi-tiered artificial neural networks has been around for a few decades already, the recent surge in computing power has enabled these conceptual models to be actually built large enough to explore their full potential.
Next to fully connected neural networks, there are a few other types of networks of interest that I will explore. One such is the Convolutional Neural Network which I will describe in more detail in section 5.1. This

type especially has benefited from advances in technology because of their highly parallel nature which effectively allows them to be run on consumer graphic hardware which is ever-improving and becoming more accessible.

The areas that have benefited most from this branch of machine learning include without doubt image recognition and speech recognition, both problems with extremely noisy real-world data that were typically tackled by hand-crafting higher level features and learning from those, yet it is far from limited to these. Deep learning has been extensively and successfully applied to reinforcement learning as well, the main topic of interest in this thesis.

The rest of this chapter will start by introducing core deep learning concepts, then proceed to applying deep learning to the reinforcement learning case. Sections 5.1 and 5.3 concern themselves with general deep learning techniques, i.e. not specific to the reinforcement learning. Section 5.4 explores the Deep Q-Network architecture which forms the basis for the rest of this thesis.

5.1 Convolutional Neural Networks

The core to deep learning is that representations of data can be learned in an unsupervised or semi-supervised manner without the need for hand-crafted features. Convolutional networks are able to learn features from raw input without any human intervention. Stacking convolutional layers even allows hierarchical features to be learned. This way the initial layer could function as an edge detector whereas deeper layers would learn higher-level abstractions such as for example facial features in the case of image recognition.
Another core property of convolutional layers is that, unlike a regular fully connected neural network, it can recognize the same features regardless of the location of the feature in the input. A regular fully connected network would need to have its weights in each of the possible locations trained to recognize the same feature which is wasteful in both space and computational requirements.

Convolutional networks are strongly inspired by the animal visual cortex which contains two basic cell types. Simple cells activate respond most strongly to edge patterns; they function as edge-detectors and have a small

receptive field. Complex cells on the other hands have a larger receptive field and are also spatially invariant to pattern location.
An early predecessor of the convolutional network that tried to capture these concepts is the neocognitron (Fukushima, 1980) which differs mostly in that convolutional networks share weights across several positions in the input, as I will explain further down. The architecture I will explain in the following section is based on the famous LeNet-5, designed by Yann LeCun, Bottou, Bengio, and Haffner (1998) to successfully recognize hand-written letters.

5.2 Building Blocks

5.2.1 Convolutional Layer

The core building block for a convolutional network is the convolutional layer. For the remainder of this text it is easiest to think of input as images, possibly with a depth dimension (such as color).

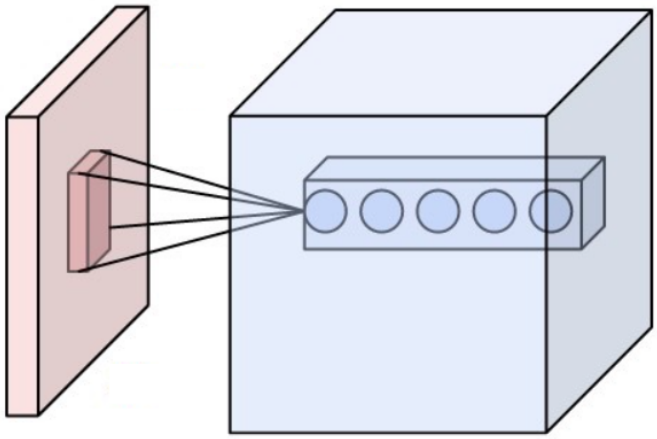


Figure 5.1: Convolutional layer (blue) connected to an input layer. Highlighted are multiple filters connecting to a single input tile.

Let us start with the notion of a single, trainable neuron in the convolutional layer. It connects to only a small part of the underlying input layer as demonstrated in Figure 5.1, albeit through the whole depth. This is the receptive field as I described earlier. Strongly connecting to only a small

part of the input is what allows convolutional networks to exploit strong local correlations that are inherently present in natural images but also in many other forms of naturally occurring data.

This single neuron is part of a filter that spans the whole input image, it has neighbors that just like it fully connect to neighboring patches of input. One such a filter is now a flat 2D activation map for a single feature. This replication for a single filter is what makes convolutional networks insensitive to the location of a feature.
Of course we would like to have multiple features, so we stack multiple of these filters on top of one another. This stacking behavior of filters is described in Figure 5.1, where a single input patch is shown to correspond to several filters.

5.2.2 Parameter Sharing

As stated before, one feature corresponds to neurons across the whole input by each corresponding to some patch of it. This relies on the assumption that the feature could arise anywhere in the input and would be useful to discover anywhere. In order to actually compute the same feature, we constrain the weights and bias of the neurons for a single feature to be shared.

5.2.3 Details

To actually generate the feature map we *convolve* the input with a linear filter, add a bias and afterwards apply a non-linear function such as a rectifier (described in Section 3.3). It is only because the weights are shared across a single feature that this convolution is possible, hence the name of the layer.

A single feature map k in terms of its input x could be computed as:

$$h_k = tanh((W_k \cdot x) + b_k) \tag{5.1}$$

It is also this operation that allows convolutional networks to run so efficiently on parallel hardware, making it the ideal candidate for general purpose GPU computing.

5.2.4 Tuning

A single convolutional layer has some hyperparameters that are both important and hard to tweak. Since learning a convolutional network is still a rather slow endeavor, it is best to start out with good estimates from the deep learning community.

The common way to build a convolutional network from convolutional layers is to have the layers closer to the input compute few features, computed over large receptive fields or tiles. As the network grows deeper, inputs to layers represent higher level features and can thus be learned from smaller input tiles; intuitively, a few high-level features can contain the same information of more lower-level features. Conversely, while there are only few worthwhile raw features such as different types of edges, there are probably more distinct higher-level features that can be used to generate the final output. Deeper layers should thus grow deeper yet slimmer spatially. This setup typically results in a funnel-like structure such as shown in Figure 5.2. The deepening occurs because of the increase in features whereas the slimming usually occurs because multiple outputs from an input layer correspond to only a single neuron, spatially, in the next layer. However, there are ways to train a neuron on a patch of input yet still retain output size (again, spatially). These I will describe below.

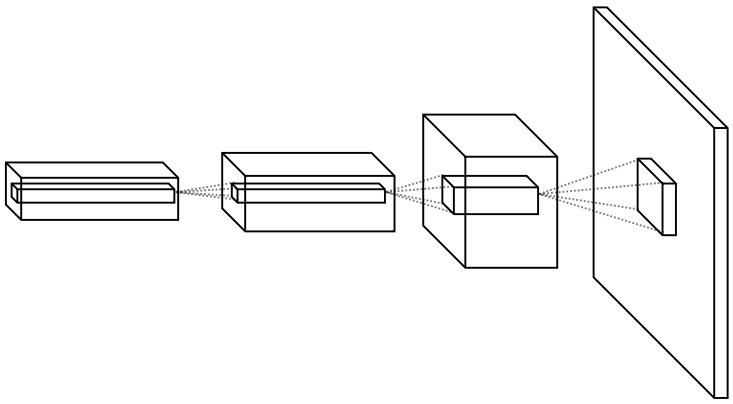


Figure 5.2: Convolutional layers can be stacked in a funnel-like manner (right to left), growing smaller spatially yet larger in the depth or feature dimension.

As noted, both the amount of features as well as the size of the receptive field in a layer play a huge role in the representational capacity of the layer.

Still, there are other ways to affect this.

Stride The stride for a convolutional layer determines the spacing between receptive fields, or tiles. A stride of 1 would have very strongly overlapping tiles and as a result larger spatial dimensions than a lower stride would have. A stride equal to the tile size would result in non-overlapping but touching tiles. An even larger one would, of course, result in unused input elements and is therefore rather unpopular in practice.

Padding If not all input elements can be used because of the filter size or the combination of filter size and stride, one can choose to pad the input with zeroes to achieve a valid convolution.

5.2.5 Pooling Layer

An important yet simpler concept to convolutional networks is pooling, a non-linear downsampling of the input. The most popular form is max pooling, demonstrated in Figure 5.3.

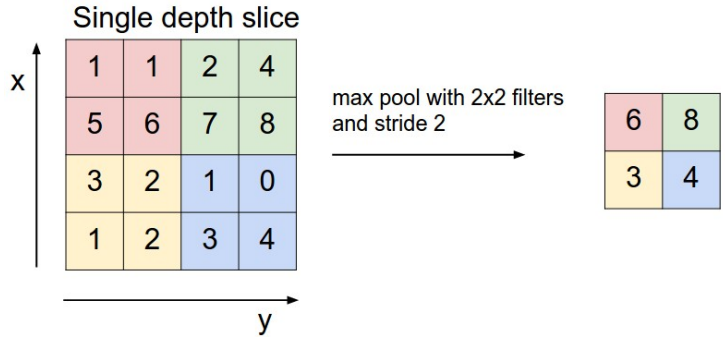


Figure 5.3: Max pooling with a filter size of 2×2 and stride 2.

Pooling reduces the spatial size while retaining the depth as the depth describes the amount of features. The idea behind the concept is that spatial location of a feature is less important than the activation of the feature (note again the importance of translation invariance), along with the notion that features are most important in relation to other features. Pooling therefore preserves relative location of each feature.

Even though most convolutional networks tended to have pooling layers, if not occasionally in between convolutional layers then often right after them, in recent years in the literature there has been a tendency to avoid pooling layers altogether. Springenberg, Dosovitskiy, Brox, and Riedmiller (2015) suggest that pooling does not always improve performance if the network already has enough capacity for the data at hand and indeed advocate the use of convolutional layers with larger strides and even more convolutional layer to make up for the loss in power.

5.3 Recurrent Neural Networks

Some tasks are sequential in nature, meaning one sample depends on a previous one, rather than data being independently drawn from some underlying distribution. Typical problems include speech and written language. Recurrent neural networks are especially suited for these domains. They process input sequences one step at a time, maintaining hidden state which will then affect future output. Recurrent networks are achieved by introducing connections that form cycles.

It is no wonder this class of artificial neural networks draws the eye of the reinforcement learner designer. A problem with a pure Markov state space has no need of a network useful for learning dependencies between different states, since by definition every state on its own is sufficient enough of a representation and includes all necessary past states in its description. However, some conceivable reinforcement learning problems are not Markovian or are only made to be so by supplying extra information to the system such as derived state (imagine angular velocity in the cart pole experiment). Both kinds of problems could still benefit from RNNs.
Recall also from section 4.3.2 the class of Partially Observable Markov Decision Processes (POMDP). These are processes that simply lack all required information in at least some state descriptions. As a result this is the class of problems that benefit most from recurrent neural networks and as a result are of special interest in this thesis.

5.3.1 Building Blocks and Training

As Recurrent Neural Networks form a class of networks, I will go over the general approach taken instead of a specific version.

Whereas regular feed-forward networks contain only forward connections, RNNs usually contain cycles allowing them to capture time dependencies. The cycle allows on state to depend on a previous one, making it the ideal setup for sequences.

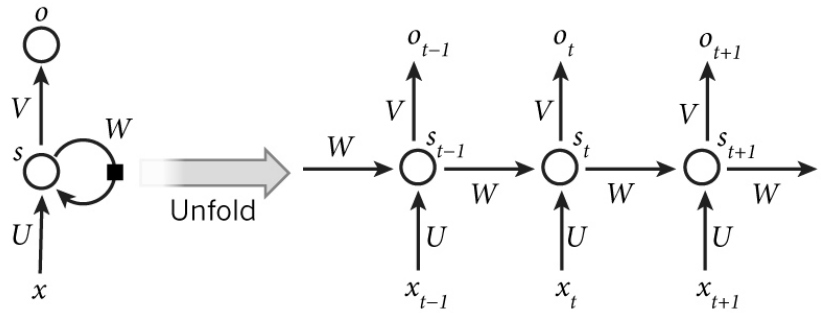


Figure 5.4: Recurrent Neural Network unfolded through time (Lecun, Bengio, & Hinton, 2015).

At first, it might look daunting to train a network with recurrent connections and as a result a recursive component in the error. However, recurrent neural networks can simply be trained with gradient descent. Figure 5.4 shows how a recurrent connection can be viewed throughout three different time steps. Seen in this rolled out fashion, it simply becomes a very deep network with shared weights and allows us to apply the well-known backpropagation algorithm.

5.3.2 Long Short-Term Memory Networks

Regular recurrent neural networks can suffer from two problems during training: *vanishing gradient* and exploding gradient. Highlighted first by Hochreiter (1991) and further characterized by **bengio1994** (**bengio1994**), these two recurring problematic phenomena have long prevented efficient training of recurrent neural networks.
The *exploding gradient* describes the norm of the gradient increasing during training because of explosive growth of the long-term components which then far outweigh the short-term components. The opposite phenomenon, *vanishing gradient*, features more prominently in the literature. It describes long term components that go exponentially fast to zero, making it practically impossible to learn long-term time dependencies of arbitrary length.

To deal with the exploding gradient, (Pascanu, Mikolov, & Bengio, 2012) suggest clipping the norm of the gradients and Graves (2013) show that *skip connections*, i.e. connections that 'skip' a layer, help mitigate the vanishing gradient problem for deep networks.

An architecture especially good at storing and accessing information is the *Long Short-Term Memory*, introduced by Hochreiter and Urgen Schmidhuber (1997). It remedies extreme gradients by enforcing a constant error flow through special internal units. It also contains gates that regulate which inputs get remembered or even forgotten, as well as gates that regulate when to output a remembered value.

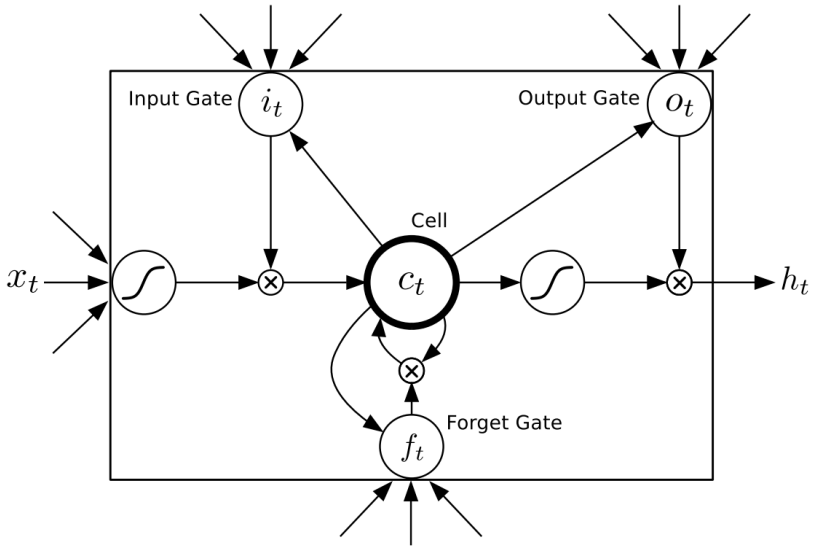


Figure 5.5: Long Short-Term Memory cell (Graves, 2013).

The basic unit is a Long Short-Term Memory cell which is displayed schematically in Figure 5.5. A cell contains three gates: an input gate, forget gate and output gate. Each gate uses an activation function, often the sigmoid function. Central to it all is the cell unit, the internal state, which gets regulated by the gates in a fashion described by their names; one gate controls whether the internal state should be overridden, the other whether it should be used in the output and yet another whether the internal state should be forgotten. The combination of these elements make the LSTM cell into into a veritable piece of memory.

Quite a lot of LSTM variants have been developed over the years since the LSTM's inception. Below I describe the relations between the different components as they are used in this thesis, based on a popular design described by Graves (2013). Note that σ is the sigmoid activation function as described in section 3.2.2.

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \tag{5.2}$$
$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \tag{5.3}$$
$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_{t-1} + b_o) \tag{5.4}$$
$$c_t = f_t c_{t-1} + i_t tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \tag{5.5}$$
$$h_t = o_t tanh(c_t) \tag{5.6}$$

The W_{ij} notation denotes the weights associated with the connection from unit i to unit j. Likewise, b_k denotes the bias for unit k.
While the equations might look daunting at first glance, they can all be given intuitive meaning. The three gates behave similarly; they all depend on the current cell input (x_t), the previous output (h_{t-1}) and previous internal cell state (c_{t-1}). All combine these inputs using a linear computation and become non-linear through the activation function. The connections from the previous cell state to a gate are called *peephole connections*, first introduced by **gers2000recurrent** (**gers2000recurrent**).

The internal cell state is a combination of a 'forget component' that determines how the previous cell state is carried over along with a combination of the current input and the previous output, weighted by the current input gate which regulates how the input should be used.
It must be noted that the original LSTM proposal did not contain a forget gate and simply added unchanged cell state back into the current update. This cell state was then referred to as the Constant Error Carousel (CEC), named so because it enforced a constant error in order to mitigate the vanishing and exploding gradient problems.

5.4 Deep Q-Network

In 2013, Mnih et al. at DeepMind published a network with raw pixels as input that successfully approached and even outperformed human ability on a range of Atari 2600 games. Two years later, in 2015, Mnih et al.

perfected the design of the network which allowed it to be on par and often outperform human players on the majority of games considered.
Ever since, extensions and alternatives based on the same principles have received a lot of attention. One such a recent alternative is Deep DPG (Lillicrap et al., 2016) which is an actor-critic approach which learns a parameterized policy instead of a value function. Interesting extensions include Prioritized Experience Replay (Schaul, Quan, Antonoglou, Silver, & Deepmind, 2016) which speeds up learning significantly by making smarter use of stored experience and Deep Recurrent Q-Network (Hausknecht & Stone, 2015) which incorporates a recurrent layer into its architecture.

This section is devoted to explaining why the Deep Q-Network performed so well and what set it apart from previous attempts. While there are two publications on the DQN architecture (2013 and 2015), the core as I explain it here is roughly the same; they differ on a wide array of network parameters and learning parameters, yet there are few structural differences. When I use DQN in the text to follow, I refer to the conceptual architecture that underlies both except where noted explicitly.

5.4.1 Problem Setting

The problem DQN attempts to solve is that of a learning agent which finds itself playing an Atari game. It does not know which game; all it knows to be available to it are consecutive observations of raw pixel values and a scalar reward signal, along with a number of unidentified actions it can take. A more detailed investigation of this environment can be found in section 6.2 where I describe the Arcade Learning Environment.

There are quite a few things that make the environment challenging to a learning agent. While the action space is small, the state space is huge even though it is still finite. This means tabulation variants of learning algorithms are no longer feasible and function approximation must be employed instead. This approximator used is a complex network, the architecture of which is described further down.

Then there are a few difficulties unique to the reinforcement learning problem.
The first is the *credit assignment problem*: given a reward, how should the learner distribute credit to each of its previous actions in a way that optimally reflects the impact of that action on the outcome. Credit assignment

is necessary and important in order to distinguish between truly important actions - game changers in this scenario maybe even - and less important actions that necessarily bridge the gap between crucial events.
The environment presented here features this problem in varying degrees depending on the game in question since some games can have a huge delay between the reward and the core action or sequence of actions that inevitably caused it. The fact that this delay is very game-dependent forms another challenge: we want to be able to face all games with a uniform architecture.

Another issue is the need for good function approximation while most techniques are tailored to the supervised learning setting where data comes from a static distribution. Since we update Q-values as we go and indeed bootstrap updates with old values, our target distribution is far from static and is only expected to become stable as the network converges.
Added to this is the fact that consecutive states are highly correlated, another characteristic that must be taken into account when using learning algorithms that assume independent data samples.

5.4.2 Deep Q-Learning

To cope with the various problems presented above, a learning algorithm was devised that beautifully combines and improves on previous attempts to improve learning.
To start with, while reinforcement learning is inherently online, a technique called *experience replay* (L.-J. Lin, 1993) is employed that allows us to combat some of the previously mentioned challenges. An agent's experiences are stored in a data-set $\mathcal{D}$ as quadruples $e_t = (s_t, a_t, r_t, s_{t+1})$ at each time step t. During the agent's experience gathering, every few steps, we apply a Q-learning mini-batch update to a sample of the experience $e \sim \mathcal{D}$ that is randomly drawn from the experience database.
Note that the use of experience replay condemns us to off-policy learning because the policy has already changed at the time a sample is sampled. The off-policy learning algorithm of choice for DQN was Q-learning, resulting in the complete *Deep Q-Learning* algorithm in listing 2.

Another important addition to the algorithm is the use of a separate network to generate target values for Q-value updates. Say the network is

Q, then at startup and every C updates we clone Q to obtain $\hat{Q}$, the target network that we will use to generate target Q-values for the next C updates.
This effectively makes the target distribution stable for C updates at a time. Of course it still needs to change but the frequency at which this happens makes divergence unlikely. An empirical indication of this behavior is presented in Figure B.2.

Algorithm 2 Deep Q-Learning with Experience Replay

Initialize replay memory $\mathcal{D}$ with capacity N
Initialize action-value function Q with random weights θ
Initialize $\hat{Q}$ with parameters θ' to a clone of Q
for all episode **do**
  Initialize start state s_1 and processed observation $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With chance ϵ select random action a_t
    Otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Perform action a_t, receive reward r_t and observe s_{t+1}
    Store $(\phi_t, a_t, r_t, \phi(s_{t+1}))$ in $\mathcal{D}$
    Sample random mini-batch $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    if episode terminates at $j + 1$ **then**
      $y_j \leftarrow r_j$
    else
      $y_j \leftarrow r_j + \gamma max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta')$
    end if
    Perform gradient descent step with error $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ w.r.t to the network parameters θ
    Reset $\hat{Q} = Q$ every C steps
  end for
end for

Note that the replay memory is usually finite for practical reasons. Different replacement algorithms can be used once it is full but the standard way to go about it would be to replace the oldest first in a First In First Out approach.

The advantages of the Deep Q-Learning approach described here can be summed up in the following points.

- The random draw is intended to break up the correlation between consecutive samples. This reduces the variance between updates.
- Each experience can be reused in multiple updates which boosts efficiency because neural networks tend to learn slowly and in the supervised learning context often need multiple passes over the same data-set.

  This idea of experience reuse has been further capitalized on by Schaul et al. who introduced *prioritized replay* (2016) based on the earlier *prioritized sweeping* (Moore & Atkeson, 1993) which prioritizes state updates according to associated change in value.
- Learning off-policy like this approach does avoids the pitfalls of self-reinforcing feedback loops where a training distribution can contain mostly samples that contain the maximizing action, a situation which would only propagate itself using on-policy learning. Such a loop is of course nefarious to exploration and can cause the learning to get stuck in a local minimum.
- Using a target network makes the learning process much more stable.

5.4.3 Model Architecture

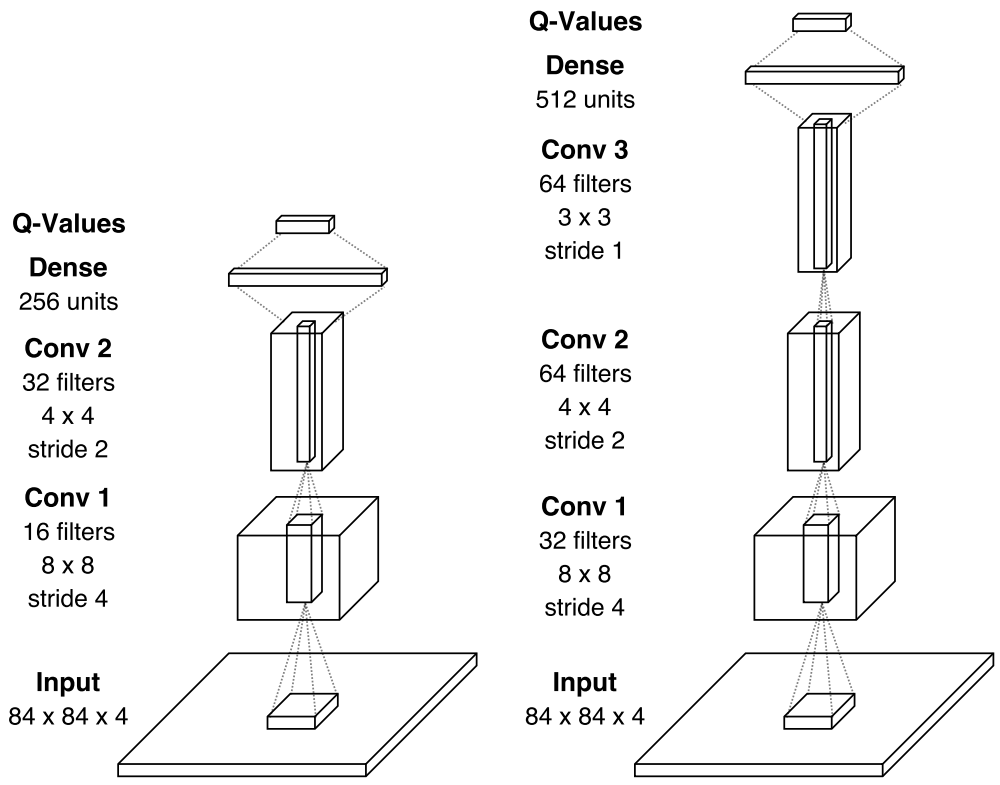
Let us start by defining the input and output, as DQN aims to be completely end-to-end. There is still a preprocessing step to make training the network more computationally feasible. Atari frames are 210 x 160 pixels in size with 7 bits for colors. To vastly decrease the input space, frames are processed to be 84 x 84. The earlier paper simply cropped the image to a region of the required size, however the viability of this approach is very game-dependent. The later version scaled the image to the appropriate size.
Even though Atari colors nicely fit into a byte, colors are converted to grayscale.
In order to make games Markovian or at least approximately so, we use 4 image frames at a time, which means two consecutive states actually have 3 frames in common. This proves to be sufficient for an agent to infer hidden features like object velocities by inspecting the distance and direction an object has traveled. In next chapter I will go into a deeper discussion on this topic.

The processing step described here corresponds to the function ϕ in listing 2. Input to the network is now of size 84 x 84 x 4. A 2D convolutional network can only cope with three dimensions: two of which spatial and one for image depth, or channels (an RGB image would have three). This is interesting to note because the depth dimension gets treated differently. To illustrate without digging too much into the details, number of image channels has no impact on the output shape of the first convolutional layer as the new depth dimension size of said output corresponds to the number of filters in the layer. After the first convolutional layer, there is nothing reminiscent in terms of shape of the input depth. Still, we have four frames at a time that we need to feed to the network. DQN handles this by having only a single image channel since everything is grayscale and treats the time dimension as the image's channels.

Usually, when designing the Q function, it would take as input a state and an action and produce the corresponding Q-value. In other words, it would have the signature $\mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$. To get the action with the highest associated Q-value for a certain state, this function would then have to be evaluated for each possible action. While we only have a maximum of 18 actions, the DQN approach is considerably more efficient by computing the Q-values for all actions in a single forward pass. The signature is now $\mathcal{S} \mapsto \mathcal{A} \times \mathbb{R}$. This means that the network will have an output unit for each action whose value is the associated Q-value.

Now that we have a description of both the input and output of the network, we can start the discussion on the layers in between. The two DQN architectures depicted in Figure 6.1 both follow the same concept of stacking convolutional layers that contain progressively more filters. The first layer has the least amount of filters but uses the largest tiles because it is meant for rougher low-level features, of which there are suspected to be fewer than high-level ones. The stride for each convolutional layer is set to half the tile size for the first two layers, so every tile overlaps up to four other tiles. The overlap makes it so each pixel can have an impact on multiple output units, spatially.
The output of each convolutional layer passes through a rectifier nonlinearity (Nair & Hinton, 2010).
After the final convolutional layer a single hidden fully connected layer is employed which is meant to take full use of the features distilled in the underlying convolutional layers. The output layer is another fully connected

layer with the output of each unit corresponding to the q-value of an action along with the input state.
There are a few differences between the two networks depicted in Figure 6.1. They all boil down to the same thing though: representational power. The network in Figure 5.6b has an extra convolutional layer and each layer has more units than the corresponding one in Figure 5.6a. More units mean the network has more raw capacity but might take longer to train.



(a) Network described in Mnih et al., 2013

(b) Network described in Mnih et al., 2015

Figure 5.6: Two variations of the Deep Q-Network. Both follow exactly the same principle of stacking convolutional layers that learn progressively more abstract features. The inner area that is highlighted in each of the convolutional layers illustrates how a single input tile is funneled through the network and gradually has more features learned on it, demonstrated by the layers' increasing heights.

CHAPTER

6

ARCHITECTURE EVALUATION

6.1 Problem Statement

In the Deep Q-Network section (5.4) I described an approach to deep reinforcement learning. While the approach works well, it has not managed to beat all games under consideration nor would it generalize well to just any game that fits the input description. Much care has been taken to make DQN as generic as possible, yet one detail stands out.
A single state consists of four consecutive image frames in order to make the problem more Markovian, a useful trait because it allows us to rely on useful theoretical properties that have been established for Markov settings. However, not all games carry sufficient information in the last four frames alone. Some need a few more, while others could conceivably show information that will then be needed to act correctly after a long period of time has passed. In other words, there could be hidden state that the agent nevertheless had access to at some point in the past. A good example of such a game is Blackjack where cards used in previous rounds affect the odds for later rounds.

My goal is now to explore different alternatives to this approach, to explore different ways of dealing with time. I will explore two approaches that still make use of a fixed time window like DQN uses - though they do so differently - and one without window at all that is more apt at learning long-term dependencies.

The rest of this chapter is organized as follows. First, I will examine the Arcade Learning Environment in close detail in so far as this benefits the following sections. Then, I will closely investigate the different approaches to the time problem. I start out by examining whether the 4 stacked frames approach taken by DQN is actually beneficial to the learning process in section 6.3.
The three alternatives I will discuss afterwards are *Late Fusion* (6.4), *3D Convolutions* (6.5), and *LSTMs* (6.6). Each in turn will be discussed, then explored and evaluated. Finally, an overview and comparison can be found in section 6.7.

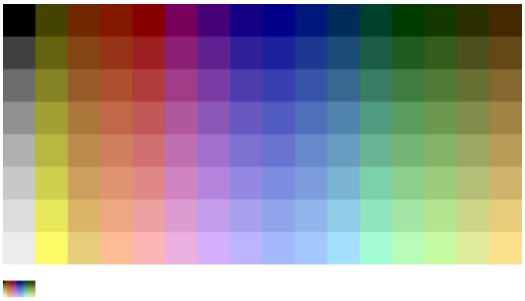
6.2 Arcade Learning Environment

In order to fully understand the experiments that follow and the implications of their results, it is important to first have a closer look at the Arcade Learning Environment (Bellemare, Naddaf, Veness, & Bowling, 2013). ALE is built on top of Stella [1], an open-source Atari emulator. It enables one to programmatically interact with any compatible Atari game file. This includes getting the screen in raw pixels, access to the Atari's emulated RAM, the game's current score and actually interacting with the game by sending actions.

To appreciate the setting completely we will need to investigate the hardware the games considered here used to run on. The Atari 2600 was released in 1977. Its CPU ran at 1.19 Mhz, more than a 1000 times slower than a single average core used for personal computers nowadays. The size of the RAM was especially small compared to what is used in modern days, with 128 bytes.
Of special interest to us is the console's graphic component. The screen is 210 by 160 pixels and could display up to 128 colors.

[1] http://stella.sourceforge.net

The Atari 2600 console is interacted with using a joystick and a button. The joystick can move in 8 directions. Combining that, along with pushing the button while moving the joystick (or not moving it) gets us to 17 distinguishable actions. Add one for not doing anything at all and we have a total of 18 actions for our learning agent.



(a) The Atari 2600 NTSC Palette used by many games. It allows 3 bytes for luminiscence and 4 bytes for chrominance, resulting in the 128 distinct colors you see here.

(b) The Atari 2600 with its joystick.

Figure 6.1

6.2.1 Shortcomings

While the Atari's computational simplicity, small screen and relatively straightforward games make it a great testbed for reinforcement learning, the same characteristics bring along a few shortcomings that bear discussing.

First off, the Atari 2600 is entirely deterministic. This allows some games to be abused by learning how to exploit bugs or otherwise causing situations that would be very hard for a human to replicate, yet that a learning algorithm could easily manage in a deterministic setting.
This exploitation of a game's weak points does not sound bad on its own - after all, the agent is learning - but it is obviously a case of overfitting which should preferably be avoided.
DQN tries to avoid this by adding a random amount of nullops at the start of the turn, that is, the agent waits a random amount of turns before it can play.

The console's determinism makes it so that given the agent's history, the next state given an action can be known exactly. However, very few games actually need more than a few frames of history in order to achieve this Markov property.
Since the main interest of this thesis is dealing with time, long-term time dependencies would be especially interesting to investigate. Sadly, however good of a testing environment ALE may seem, it lacks thoroughly in this regard. A good POMDP game is Blackjack, though this is not supported by ALE.
We will discuss an additional benchmark to circumvent this in section 6.2.3.

I will now discuss two games in some detail and shed light on their core distinctions in order to later understand and explain the differences between their learning results.

6.2.2 Space Invaders

Space Invaders is an arcade game published in 1978. The player controls a laser cannon which it can move along the bottom of the screen and to fire at aliens that move left to right and gradually downwards and can even shoot back. The player can even hide below shields which can get destroyed by lasers from either side. As the player progresses, the aliens start to move faster and faster. The game is over once they reach the bottom of the screen.

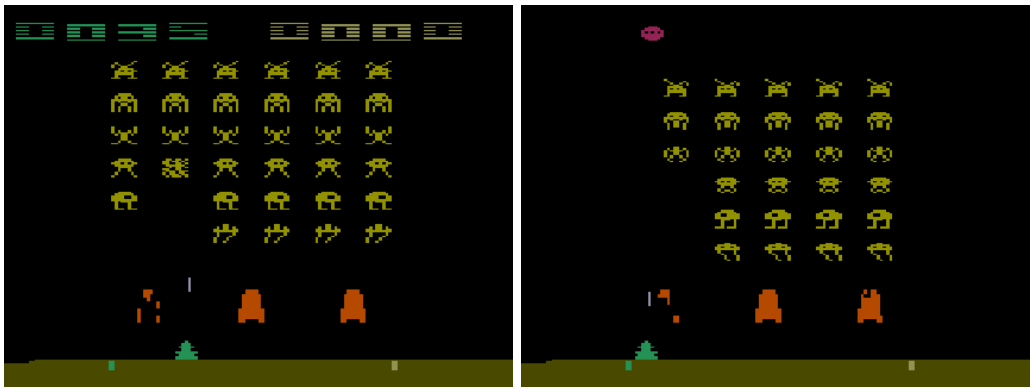


Figure 6.2: Space Invaders

Each time an alien gets killed the agent gets a reward that depends on the type. The agent needs to learn to associate this reward with the action that ultimately caused it, namely firing at the place where an alien would be at

the time the laser arrives. This can be rather tricky as aliens can move in multiple directions, so ideally the agent must somehow learn this pattern in order to excel at the game.
An agent receives a large bonus reward when it destroys a periodically appearing fast-flying saucer at the top of the screen. This event is especially hard to learn because of its rarity compared to other situations.

6.2.3 Pong

Pong was originally an arcade game by Atari. It was then ported to the Atari 2600 under the name *Video Olympics*, though in this text I will refer to it simply as Pong.

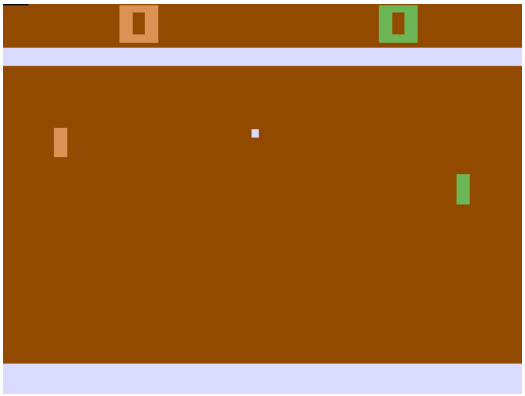


Figure 6.3: Pong

The game is a simulation of badmin. It features two paddles, each controlled by a different player (or AI). A ball is hit back and forth between the players by moving the paddle up and down and intercepting the ball to bounce it back. A point is scored when the opponent fails to do so.

Flickering Pong

Since regular Pong and indeed most Atari games become at least approximately Markovian given a small history, we would like a different benchmark to compare techniques for partially observable MDP's. Such a one has been introduced by Hausknecht and Stone (2015) with *Flickering Pong*. It is entirely the same game as Pong, except a frame can only be observed with some probability p and becomes obscured, that is, entirely black, otherwise. This results in a stochastic screen flickering for the learning agent.

In order to be successful at Flickering Pong, an agent must learn to robustly estimate variables such as ball velocity across multiple frames, half of which are expected to be obscured.

6.3 Stacked Frames Approach

In order to get a better feeling of why the dimension of time in reinforcement learning merits closer examination and further research, this section will elaborate on the current approach as described by Mnih et al. (2013). Mnih et al. found that for most games under discussion 4 frames of history is sufficient to render the game approximately or completely Markovian. This means that when considering not only a single frame but also the 3 prior to it, all information that could ever be required to make an optimal decision at that point is encompassed within this small fixed window.

In order to combine these frames, it is sufficient to make sure each image frame only has a single image channel (e.g. grayscale or luminiscence) so the image depth dimension can be reused to simply stack the frames on top of each other. This limitation is present because 2D convolutional networks can only handle three dimensions, of which over the third one (often referred to as depth or image channels) has no control in terms of filter size; it always reaches across the whole depth.
This way of stacking frames along with the rest of the standard DQN is depicted in Figure 6.4.
In order to determine whether this frame stacking approach actually benefits from the extra history, I investigated multiple the learning process for multiple games with both architectures. We are interested in the learning curve which can be characterized with total accumulated reward over time, as suggested by Bellemare et al. 2013. This is done by periodically holding greedy playthroughs for 50000 steps and then averaging the accumulated rewards over the episodes.
In Figure 6.5 we can see rewards over time as the agent progresses. While it does show general tendencies quite well, the graphs are noisy even though results are averaged over 5 runs. This is especially the case for DQN with 4 frames. As Mnih et al. (2013) note, this is due to the fact that small policy weight changes can result in vastly different visited states for the policy.

It can be often hard to tell if the agent is making steady progress using the reward curves and it can be even harder to compare different approaches.

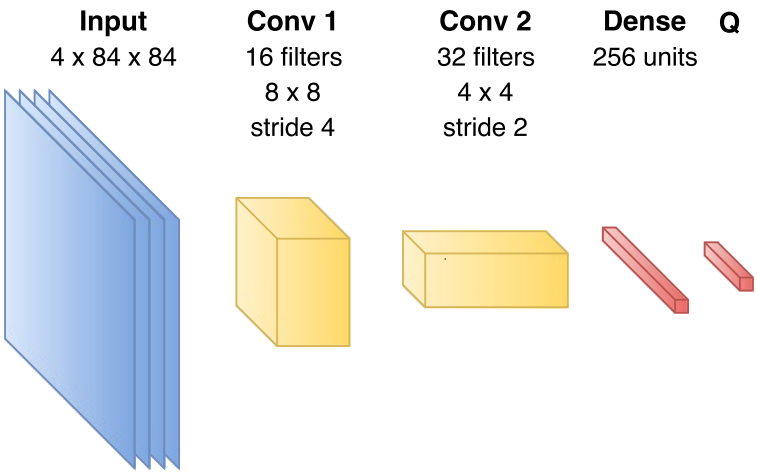


Figure 6.4: DQN architecture by Mnih et al., 2013. Note that the frames can not have any depth because of the limitations imposed by 2D convolutions.

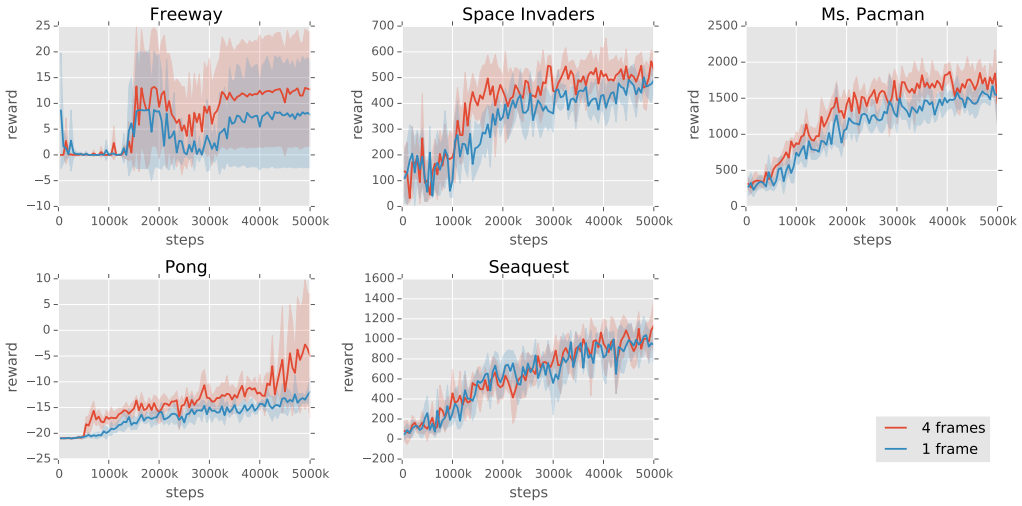


Figure 6.5: Rewards over time for 4 stacked frames as input compared to just a single one for five different games with experiment parameters as in Table A.2. Shaded areas depict the 95% confidence interval.

To combat this I will often employ an alternative method for evaluating an agent's learning behavior. At the start of a learning experiment, 3200 states are gathered using a random policy. Periodically, at the same time of the greedy playthroughs, for each of these states the maximum Q-value is computed and the average over all these states then collected as an indication of learning behavior.

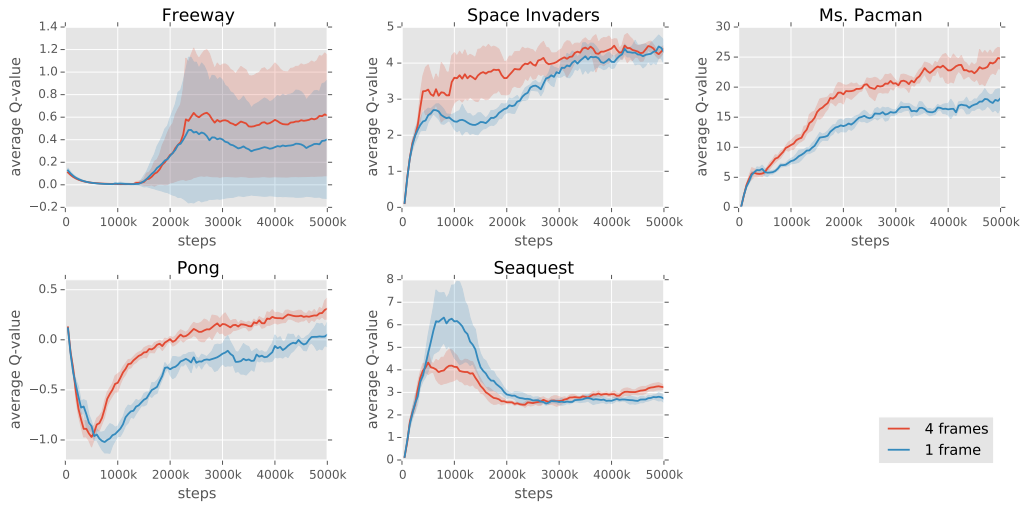


Figure 6.6: Average maximum Q-value over time for 3200 holdout states. Shaded areas depict the 95% confidence interval.

The resulting graphs in Figure 6.6 are much less noisy and give a decent indication of how well the different agents learn the games' value functions. From here on out, all experiments will be set up according to the experiment description Figure A.8.

We can now analyse the experiment's results. For 4 out 5 games we can say that the stacked frames approach performs noticeably better, although for the game Freeway results are so noisy we can not make a strong statement. The game Seaquest seems to form an outlier in that the stacked approach forms no immediately obvious advantage. It does look like it might not have stagnated entirely and is actually improving beyond the single frame DQN but we do not have data beyond that point and it has not helped within the employed 5 million training steps.

These data indicate that the stacked approach does fare better compared to just using a single frame although it is heavily dependent on each game's

specific mechanics and it is not true for any and all games. What we can not conclude with certainty however is whether the results are due to the historic information or increased capacity of the first convolutional layer in order to cope with the extra frames.

6.4 Late Fusion Network Approach

The first alternative to the standard DQN considered here is the Late Fusion DQN architecture inspired by Karpathy et al. (2014) who successfully deployed it for action recognition in video clips. The base premise for this architecture is that in order to capture time dependencies it is sufficient to look at differences between frames. Presumably, this works better with short-term dependencies as differences between frames close to each other in time are smaller.
Also, Late Fusion considers two frames with a fixed distance in time in between. This can only yield good results for longer time dependencies if the dependency always spans approximately the same amount of time, a requirement that is undesirable and not always satisfiable.

The Late Fusion DQN is based on the default DQN implementation. In fact, the learning algorithm is entirely the same and parameters are as depicted in Table A.1.
Instead of a single stack of convolutional layers, we now have two towers that each take as input a single frame. The frames are separated by a fixed amount of frames that go unused for the pass. Each of the two towers has no access to the frame that is input to the other one. Rather, the towers get combined by a fully connected layer that is also the first of the layers that has access to both time steps.
Since each convolutional network now only has a single frame as input, we suddenly get back the image channel input dimension (depth) that was previously used to stack frames together. In order to use that dimension for time, DQN uses a grayscale which only requires a single scalar value and as such this channel dimension was implicit before. Now that we have access to an extra input dimension we can again use color channels such as RGB, which might add useful information to the model.

The general gist to the Late Fusion architecture is that the convolutional layers compute high-level features which the fully-connected layer can then compare in order to compute the correct output.

The full architecture is depicted in Figure 6.7.

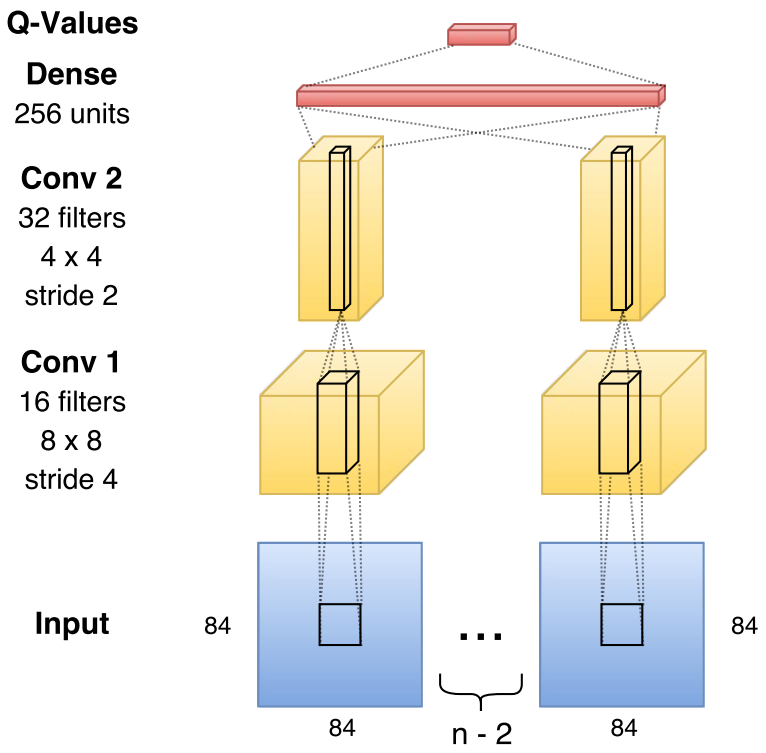


Figure 6.7: Late Fusion DQN. A sample of n consecutive frames is chosen but only the outer two frames feed into the network. The features learned by the two convolutional networks is combined through the fully-connected layer that combines both. Only from the fully-connected layer up is information from two different time steps available.

6.4.1 Tuning

We can try a few different alterations to the basic late fusion architecture. First off we can decide whether convolutional layer weights should be shared between layers at the same level or separate for each tower. Then there is the extra image dimension we have access to. In order to make use of it, I used frames with 3 color channels to hold RGB values.
A performance comparison for the game Pong is depicted in Figure 6.8.

Keeping the weights the same between convolutional layers on the same level seems to offer an advantage over separate weights. What this effec-

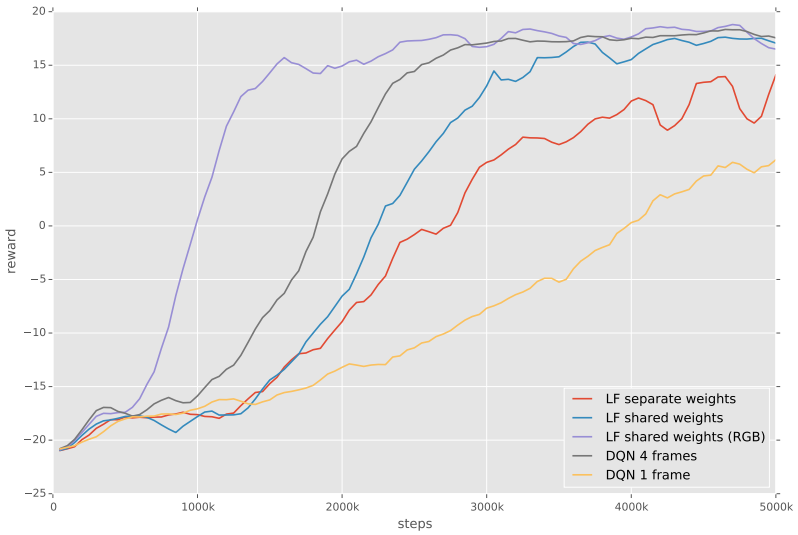


Figure 6.8: Rewards over time for Late Fusion architectures compared to standard DQN.

Architecture	Steps to 10 reward
LF shared weights	$2.6 \times 10^6 \pm 2.0 \times 10^5$
LF separate weights	$3.3 \times 10^6 \pm 5.9 \times 10^5$
LF shared weights (RGB)	$1.1 \times 10^6 \pm 7.3 \times 10^4$
DQN-4	$2.0 \times 10^6 \pm 2.9 \times 10^4$

Table 6.1: Time to threshold of an accumulated reward of 10 for Pong on Late Fusion architectures.

tively does is force the convolutional layers to learn the same features so the output of the top convolutional layers has exactly the same semantics. This way the fully-connected layer has access to the difference between frames at different time steps.
When not sharing weights each convolutional layer has to learn features independently which slows down learning and does not guarantee the same features will be learned. In fact, this approach does not even outperform the single-frame DQN.

While the shared weights network still does not learn as quickly as regular DQN, giving the network access to RGB values offers obvious gains in terms of learning speed. The motivation behind access to three separate color channels is that it is easier to learn a function on three variables than just a single one that is trying to encompass the same information. With multiple inputs (color channels) the network gains more representational power.
Figure B.1 illustrates how some games can benefit from added network capacity.

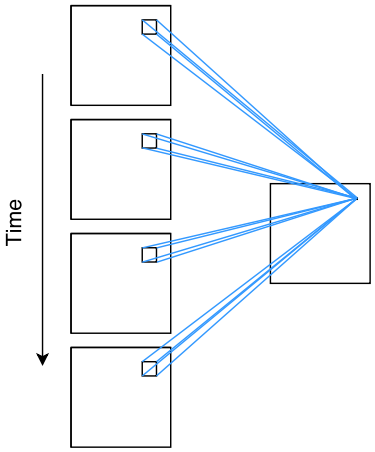
We can conclude that using two separate time steps as input to the network that are represented by the same learned features offers gains

6.5 3D Convolutional Network Approach

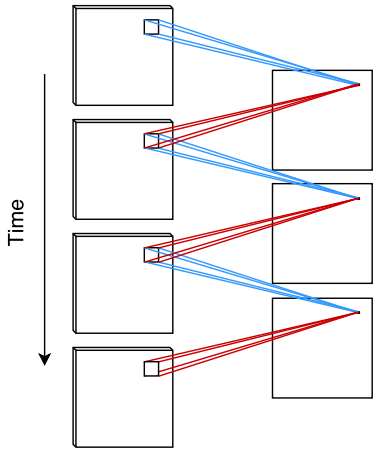
Building on the idea that time is an extra spatial dimension and should not be handled differently from image width or height, I investigate 3D convolutional layers next. The idea of convolutions over time has already been applied successfully by **claessens** (**claessens**).
In the original DQN implementation the time dimension gets flattened immediately by the first convolutional layer and is not present explicitly in its output; only the first convolutional layer has explicit access to time because the 2D convolutional layer implementation extends along the entire depth of the input. Intuitively this means that only low-level features over time can be learned whereas we can easily imagine more abstract features that relate to time. This is where the 3D convolutional network comes in, in order to allow us to specify step size and stride across time in order to control how time is propagated through the entire network.

The original 2D convolution setup in DQN is exactly the same as a 3D convolution with 4 input frames and only a single image channel (grayscale),

where the filter over time spans across all 4 input frames, as illustrated by Figure 6.9a.
A 3D convolutional layer gains an extra dimension which can be used for image depth, like colors. It also allows us to specify the filter size and stride over time. Figure 6.9b for example employs a time filter size of 2 and a stride of 1. Since this comes down to combining every consecutive pair of frames, the output of the convolutional layer has a time dimension that is shrunk by 1 compared to the number of input frames.



(a) 2D convolution for a single feature as used by Mnih et al., 2013. Since time is treated as image depth, a feature always spans across all time frames.



(b) 3D convolution for a single feature (depth slice of the convolutional layer) with 4 input frames and a time filter size of 2. Connections of the same color depict shared weights.

Figure 6.9: Convolutions over time.

The 3D Convolutional DQN is again based entirely on the original DQN implementation with parameters in table A.1.
An example instantiation of the 3D convolutional DQN can be found in Figure 6.10. It leaves the amount if input channels unspecified; in our case this can either be 1 for grayscale or 3 for RGB values. There are again two convolutional layers. Since these make use of 3D convolutions, filters over time can be specified. Both layers in Figure 6.10 use a time filter size of 2, so the time dimensions shrinks by 1 after each. The fully-connected layer still has a time dimension of size 2 to cope with in this example, as opposed to a flattened time dimension in the original DQN architecture.

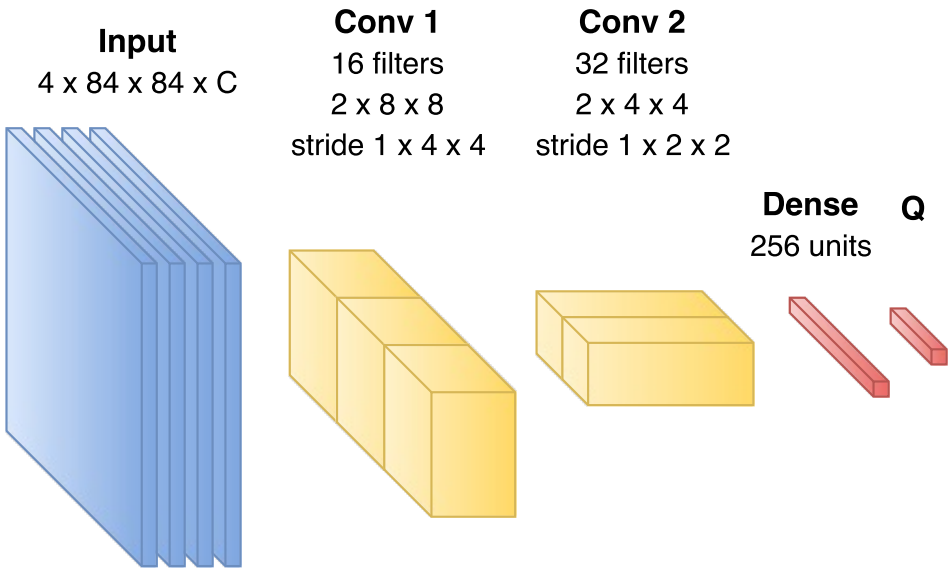


Figure 6.10: 3D convolutional DQN. The input consists of 4 frames that can each have C image channels. Both convolutional layers employ a time filter size of 2, meaning a single weight corresponds to two frames of the layer's input. Combined with a time stride of 1, the time dimension of the output of each consecutive layer is 1 smaller than its input.

6.5.1 Tuning

There are different ways to combine time frames throughout the network. A decision can be made between 'merging' (feeding multiple frames into a layer at a time) early or later on. For readability, a network with time filter sizes $n1$ and $n2$ for its two convolutional layers respectively is denoted as a $(n1, n2)$ 3D convolutional network.
Next to varying time filter sizes I also attempt a variation with an extra max pooling layer right after the second convolutional layer with a filter size of 2 across all dimensions.
Just like with the late fusion architecture a variation that uses RGB image channels has also been attempted.
The learning curves for all of these networks are shown in Figure 6.11.

(a) Reward learning curve

(b) Average Q learning curve

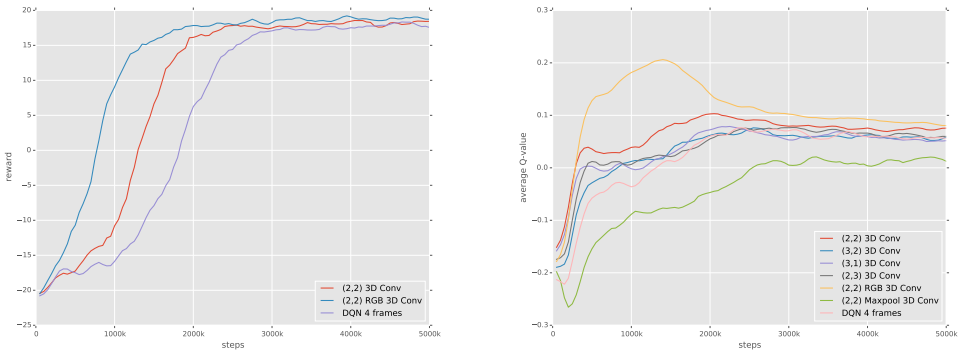


Figure 6.11: 3D convolutional architecture comparison for Pong

It can easily be seen that the max pooling variant performs worst. Not incidentally, this is the one that supplies the least information to the fully-connected layer. Among the networks with different time filter sizes, the $(2, 2)$ variant seems to learn the fastest, followed by the $(3, 1)$ variant. These are the ones that significantly outperform DQN, along with the RGB variant.
The two that perform worse and indeed are closest in performance to the standard DQN are also the ones that perform the most time merges; these provide less information to the fully-connected layer or at least provide information in a more compact manner since their output is smallest.

Architecture	Steps to 10 reward
(2,2) 3D Conv	$1.6 \times 10^6 \pm 6.0 \times 10^4$
(2,2) RGB 3D Conv	$1.0 \times 10^6 \pm 8.3 \times 10^4$
DQN-4	$2.0 \times 10^6 \pm 2.9 \times 10^4$

Table 6.2: Time to threshold of an accumulated reward of 10 for Pong on 3D convolutional architectures. The max pooling architecture is left out because it did not always manage the threshold within the allotted time.

Just as in the late fusion scenario, using RGB-colored input performs noticeably better and even fares better than standard DQN.

Another way to characterize learning speed is *time to threshold* reward. Table 6.2 shows this value for all 3D convolutional networks for a threshold reward of 10 which can be considered more than decent as it is the point at which the agent beats the AI 75% of the times. The time to threshold metric confirms the above deductions made from Figure 6.11. Statistical tests to back up conclusions can be found in Table A.4.

6.6 Long Short-Term Memory Approach

The last approach I consider involves the use of Long Short-Term Memory, a specific kind of recurrent neural network that is capable of learning time dependencies over arbitrary and unknown amounts of time (section 5.3.2). While I use LSTMs as part of a deeper neural network, LSTM units have already been successfully applied to simpler problems (Bakker, 2001).
The very same architecture I will discuss here has been attempted by Hausknecht and Stone (2015) who found favorable results for some few games. Hausknecht and Stone concluded that few significant differences were found with the original DQN architecture and that no general benefit could be gained compared to simply stacking frames. One could however argue that merely getting rid of the fixed window which stacking automatically implies presents a merit in its own right and allows for better generalization of the architecture to other problem domains.

Recall that the LSTM unit as we consider it here contains three *gates* and an internal cell state (Section 5.3.2). The input gate and forget gate together

govern the content of the internal state, which forms the memorizing core of the unit, whereas the output gate learns how to output the internal state in order to create the gate's output.

The intuitive idea behind the use of LSTMs is that the units will learn to remember important events and then use that information at some later time when it is required. Given their goal, they are especially useful in Partially Observable Markov Decision Processes such as we have when considering only one Atari frame at a time. Sadly, Atari games generally only have very short time dependencies which make them an imperfect testbed for long-term dependency learning so we will focus on learning those instead.

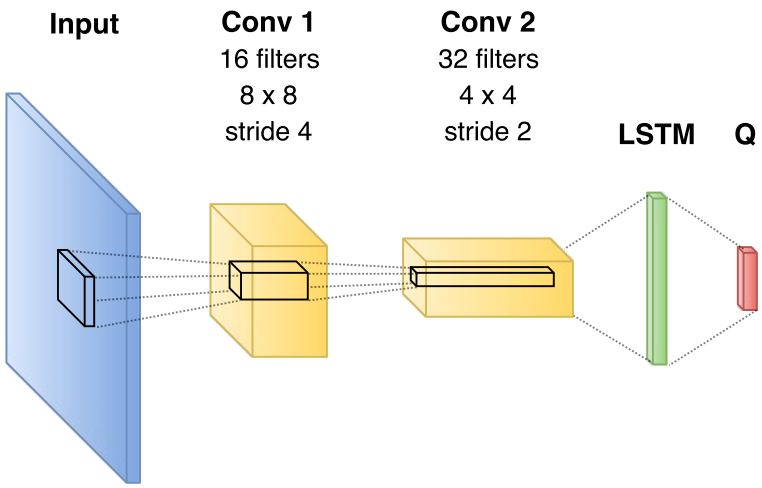


Figure 6.12: LSTM DQN

6.6.1 Design and Training

The Long Short-Term Memory variant of the deep Q-network - or Deep Recurrent Q-Network, as named by Hausknecht and Stone, 2015 - is once again based on the DQN version by Mnih et al., 2013, with common parameters in Table A.1 and LSTM-specific parameters in Table A.3.

The network is constructed by taking out the first fully-connected layer and replacing it with an LSTM layer with the same or different amount of units. Everything else except the input stays the same. Instead of stacking frames along the depth dimension of the input, only one frame at a time now gets

fed to the network. It is up the LSTM layer to handle memorization across time. Just like with the previous two approaches, ridding ourselves of the stacked frames opens up the input depth dimension once again to be used for image channels like RGB values.
The network architecture is displayed in Figure 6.12.

In the original Deep Q-Learning algorithm, after each action selection a mini-batch learning update would occur. LSTMs require input frames to be observed in chronological order, however. This learning approach would not work because data gathering steps would be interleaved with mini-batch updates, breaking the required flow of time, so the LSTM units would consider the mini-batch to follow the observed frame right before the update which is of course not the case. On top of that, mini-batches contain random samples in order to break correlation between consecutive frames while LSTMs, again, require uninterrupted sequences.

In order to accomodate the new requirements, I adapted the Deep Q-Learning algorithm to keep sequential information intact at all times. First off, this required a new sampling method. Instead of sampling mini-batches of 32 random samples, I now sample *random sequences*. This is done by selecting a single random time step, then including consecutive time steps until either the batch size is reached or the episode terminates. In the following experiments, batch size is always equal to the amount of unroll steps: this is the number of steps included in the backpropagated gradient. The batch then gets treated like it would be normally, feeding it into the learning update while keeping the order intact so the LSTM's internal state can follow along.

This brings me to the internal state reset and backup. At the start of any episode, the LSTM's internal state along with its output history gets reset to initial values. These initial values can be fixed and supplied by the network designer or even learned dynamically. The reset is required to make clear to the LSTM that the first state of an episode does not follow the terminal state of the last episode.
A parameter *backup* and subsequent reset on the other hand occur at the start of every mini-batch learning update. This is done so the sequential mini-batch forms an isolated sequence of events and no accidental information from before the update gets carried into the update by the LSTM's internal memory. After the update the internal state and output history are

restored from the backup so the agent can go on gathering data as if no interruption had just occurred.

6.6.2 Tuning

There are a couple of hyper parameters to an LSTM unit that one could consider exploring. I will restrict myself to two here: number of units in the layer, which effectively determines layer capacity, and number of unroll steps which affects how many time steps are included in the update.
Reward and average Q curves are depicted in Figure 6.13.

(a) Reward learning curve (b) Average Q learning curve

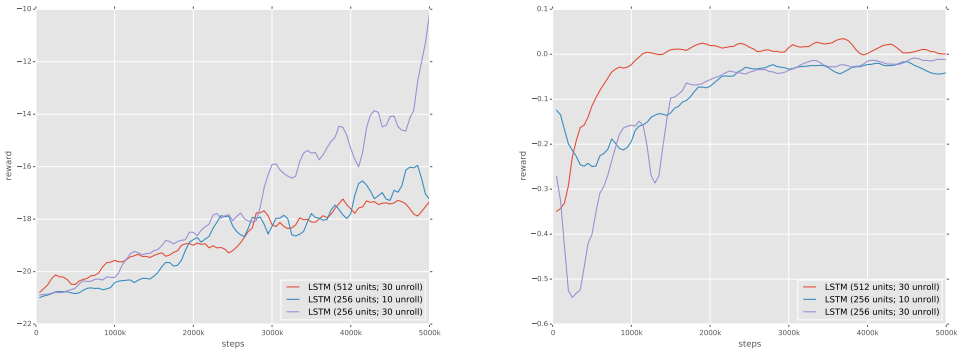


Figure 6.13: LSTM learning curves

It is hard to make out results from the reward curve because of its extreme noisiness. Interestingly, the network with fewer units and more unroll steps included in the gradient is noisiest both in its performance and in its Q learning curve, yet it is also the one that reaches highest in terms of reward. It also seems that it might still be learning at the time the experiment is stopped.
The network with more units seems to learn most stably. It gets a good start and its Q-values stay approximately stable for a long time. Yet it still does not outperform the network with radically different settings (less units and less gradient steps).

Compared relatively, the different architectures do not differ radically in terms of performance. However, they do underwhelm heavily when compared to regular DQN which is not displayed in order to not distort the

graphs too much. While the reward curves do seem to go up, they do so a lot more slowly than with the previous methods this thesis considers. Next section will form an overview of all methods.

6.7 Overview

For this section, each architecture has its hyper parameters set to the variant that performed best in each of the previous tuning sections.

Architecture	Parameters
Late Fusion	Shared parameters
3D Conv	time filters of both conv layers of size 2
LSTM	256 units, 30 rollout

Table 6.3: Best hyper parameters for each architecture insofar as these are explored in the corresponding tuning sections.

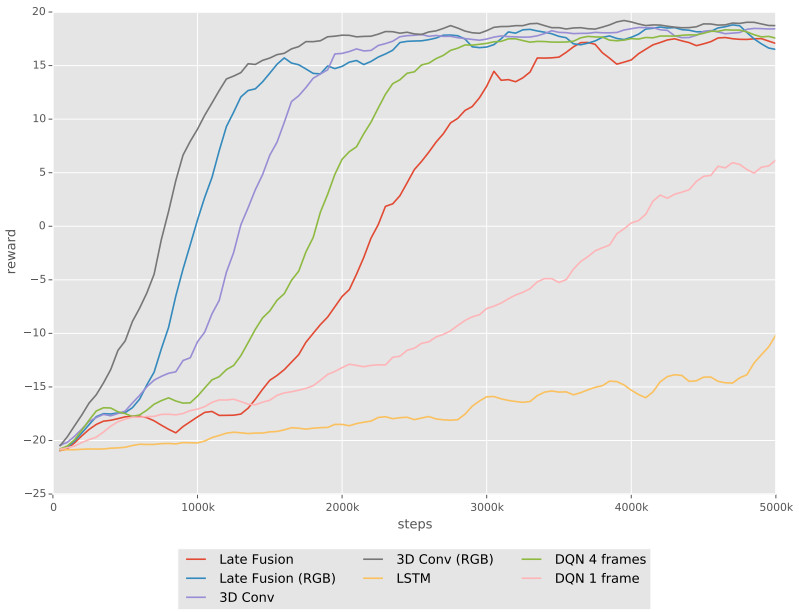


Figure 6.14: Rewards over time for the best-performing instantiations of each architecture for Pong.

Figure 6.14 offers an overview of each architecture's learning curve in terms of accumulated reward. I have included the RGB variants though as noted before there is a strong suspicion that these outperform others because of the added network capacity. Still, it is interesting to compare the two against one another.

Note that for Late Fusion and 3D Conv, the difference between their RGB variants is far smaller than the difference between their regular versions. Or, viewed otherwise, employing multiple channels gives the Late Fusion network much more of a boost in performance than it does 3D Conv. Using RGB seems to close the gap somewhat, it levels the playing field. The Late Fusion variant with RGB outperforms both regular 3D Conv and standard DQN, meaning the added information and capacity make a vast difference.

Without making use of extra input channels, the only method I considered that outperforms standard DQN is the 3D convolutional network. It is capable of learning features over space and time at each level of abstraction.

While the network that employs an LSTM layer does seem to be learning and does not look like it has finished doing so, it learns terribly slowly which makes it an infeasible approach for the problems considered here. The goal was for the LSTM units to learn short-term dependencies between frames, just like the networks that worked over multiple frames did. Sadly, the test problems at hand did not contain any long-term dependencies that could be learned to improve performance so we cannot compare LSTM to other networks on this alleged strength, only short-term dependencies.

Upon closer examination of the LSTM layer's internal state after an episode, it seems like it has shifted only small amounts from its initial zero initialization at the start of an episode. It may be the case that the LSTM's internal state struggles to keep up with the changes in its input. The way the input affects the internal state is regulated by the input gate, the training of which is in turn affected by the combined interplay with the other gates that all have to be trained themselves. It is not inconceivable that the training of this whole contraption that is the LSTM unit takes longer than it would the training of a simpler layer such as the fully-connected one in DQN.

6.7.1 Flickering Pong

The Flickering Pong experiment, introduced in section 6.2.3, is intended to obscure observed state tremendously in order to further discern which architectures perform well on a partially observable MDP. For the following analysis I ran all architectures on the Flickering Pong game with $p = 0.5$ so half of the observations are expected to be completely obscured. Each observations is stored in the replay memory as it is perceived and testing occurs in the same flickering environment.

Architectures are tested with both 4 and 10 frames as input. The LSTM architecture is left out in the following discussion because its learning progress was already terribly slow compared to other architectures. Indeed, preliminary experiments showed that the LSTM architecture barely learned anything in terms of Q-values for the duration of the experiments.

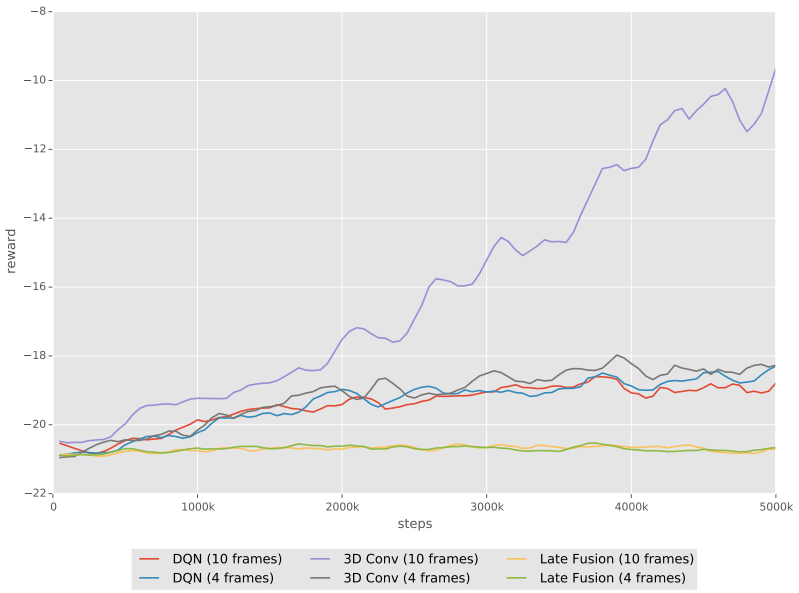


Figure 6.15: Rewards over time for the best-performing instantiations of each architecture for Flickering Pong with $p = 0.5$.

Perhaps not unexpected, Figure 6.15 demonstrates how poorly the Late Fusion network performs on the Flickering Pong game. The idea behind the architecture is that differences between two frames can be learned. In this experiment however, there is only a 25% chance that both input frames are unobscured. With no internal memory to guide the network, it cannot

be expected to learn meaningful representations over input that mostly contains blacked out frames.

The surprising result is that the 3D convolutional network given 10 frames did spectactulary better than any other approach in this comparison. From the learning curve it can not be excluded that it is even still learning while the other approaches seem to have stagnated. The fact that the network requires more than 4 frames as input to exhibit this behavior is maybe not all that surprising, given that there is still a 6.25% chance that the input consists entirely of obscured frames.

It looks like the 3D convolutional network given sufficient frames is the only architecture that keeps learning steadily from its input while others have stagnated. We expect this is due to the way features over multiple time frames are learned and time information is preserved throughout the network. Given 10 input frames, the 3D convolutional network considered here still contains a time dimension of 8 as input to the fully-connected layer as each convolutional layer merges every two adjacent frames, decreasing the time dimension by 1.

CHAPTER

7

CONCLUSIONS

The problems under consideration are games in the Arcade Learning Environment, an Atari game emulator which allows a learning agent to interact with its environment. The learning agent observes images as raw pixels and must successfully learn a value function, embodied by the network, that indicates which actions are best in which states. Along the way the network also has to learn how to extract useful features from these raw frames in a hierarchic fashion in order to do so.

The goal of this thesis was to improve the performance of the standard Deep Q-Network in the Arcade Learning Environment by exploring different approaches to learning dependencies over time.
DQN employs a straightforward frame stacking approach: frames are always fed into the network four at a time in order to render the games Markovian, meaning that any useful historic information is present in this small stack and nothing beyond could influence decisions.
We have started out by showing that this approach is already an improvement over using just a single frame because important information such as object velocity is lost, though the boost in performance is heavily dependent on the game under consideration.

We have explored three alternative approaches to learning dependencies between frames over time. The first two of these employ a fixed history to learn features that can be derived from short time spans while the latter can learn to leverage information observed an arbitrary amount of time before.

The Late Fusion network proposed here is based on the Late Fusion architecture that has already successfully been used for action recognition in video, though our variant is heavily based on DQN for better comparison. Instead of a single stack of convolutional layers, the Late Fusion architecture contains two such stacks, or towers, that each take as input a single frame though both share weights in order to learn exactly the same features. Only the fully-connected layer that both towers feed into has access to time information because it can see the difference between the high-level features from both stacks.
The Late Fusion network has a learning process slower than standard DQN but still forms an improvement over using just a single frame.

The 3D convolutional architecture can learn features over time and space combined at multiple hierarchical levels because it gives more control over how filters over time are learned and propagated throughout the network. We have shown that it significantly outperforms standard DQN. Moreover, it is shown to vastly surpass any other approach on the stochastic POMDP Flickering Pong given sufficient input frames.

Both of these architectures have access to an extra input dimension compared to DQN because of the way they handle time. While DQN converts images to grayscale before feeding them into the network, we have shown that making use of the extra dimension to hold RGB color information significantly boosts learning speed of both architectures.

Lastly, we have constructed an LSTM-based DQN architecture with the goal in mind for it to learn how to retain information from previous frames. LSTM units keep internal cell state, the use of which is regulated by an input gate, forget gate and output gate that all have to be trained. They also contain a recurring connection. This makes them ideal for learning over sequences which reinforcement learning inherently exhibit in its sequential nature.
While this approach does exhibit a learning curve, it is considerably slower than all other considered approaches.

7.1 Contributions

We have introduced the Late Fusion architecture, originally from supervised learning, in the reinforcement learning context to learn value functions.
Additionally, we have introduced a deep 3D convolutional network capable of learning features over time as alternative to the currently widely used 2D convolutional DQN.
Finally, we have investigated an LSTM-based deep network and a variant of the Deep Q-Learning algorithm tailored to learning over sequences and found it wanting in terms of learning speed on the Arcade Learning Environment.

For all of these architectures, we have considered multiple combinations of hyper parameters for the Arcade Learning Environment and tuned these to find the fastest learning variant. For the 3D convolutional network, this resulted in the network that keeps a uniform time filter size of 2, slowly merging time throughout the network.
For the LSTM-based network, results were not as clear though the network with 256 units and a gradient step of 30 fared better than both the variant with more units and the variant with a smaller gradient step.

In addition to tuning, all architectures considered here give access to an extra image dimension, the use of which we have tested with RGB colors instead of the standard grayscale. We have tried the Late Fusion and 3D convolutional networks with this enriched input and found this provided significant performance gains.

Finally, because we found the ALE benchmark wanting and needed a more clear example of a POMDP, we have used Flickering Pong, a stochastically heavily obscured variant of Pong, as an additional benchmark and found only the 3D convolutional network with sufficient frames managed to learn decent performance on this problem.

7.2 Future Work

The experiments in this thesis demonstrated how the LSTM architecture learned rather slowly compared to the other architectures considered, while

the other architectures perform well but can only learn short-term dependencies because of their use of a fixed history window. It may be interesting to combine the strengths of both approaches; one could employ the short-term approaches suggested here to learn features over small spans of time - typically these are derived features such as velocity - combined with recurrent units that operate on abstract features (such as is already attempted in this work) which includes features over short periods of time.

Additionally, we believe performance is somewhat hindered by performing a cold restart on the LSTM layer's internal cell states. It may be interesting to learn initial values for this internal state so more of the training process carries over between training episodes instead of only gate weight updates.

Appendices

APPENDIX

A

SUPPLEMENTARY TABLES

Table A.1: Basic experiment hyper parameters

Hyperparameter	Value	Description
mini-batch size	32	Number of random samples for each stochastic gradient descent update
replay memory size	10^6	Replay memory contains this number of most recent frames for SGD updates
agent history length	4	Number of most recent frames used as input to the Q network
target network update frequency	2000	Every 2000 parameter updates the target network gets updated with the parameters of the current network.
discount factor	0.95	Discount γ used in Q-learning update
frame skip	4	The agent only sees every 4th frame
update frequency	1	Parameter updates occur every action selection
learning rate	$2 \cdot 10^{-4}$	Learning rate η used by RMSProp
RMS decay	0.99	Gradient moving average decay factor ρ for RMSProp
RMS epsilon	10^{-6}	RMSProp small stability value added to denominator (see equation 3.24)
clip delta	false	Whether to clip the delta; Mnih et al., 2015 clips this to $[-1, 1]$
initial exploration	1	Initial value for ϵ in ϵ-greedy
final exploration	0.1	Smallest value for ϵ in ϵ-greedy
epsilon decay	10^{-6}	ϵ decays linearly over this amount of actions
replay start size	100	Wait with training until the replay memory contains at least this amount of frames
max no-ops	10*	Start with a random amount of no-ops that is at most this amount
resize frame method	scale**	Processing step employed to resize frames in order to reduce amount of pixels

The values in this table are based entirely on the DQN implementation by Mnih et al., 2013, except for those annotated.
* The original implementation uses no initial no-ops. This is added in to add some stochasticity to the games.
** The original implementation crops out the required area from a frame. This does not generalize well to games where important information falls outside of this frame.

Table A.2: *Boost* experiment hyper parameters

Hyperparameter	Value	Description
discount factor	0.99	Discount γ used in Q-learning update
update frequency	4	Parameter updates occur every action selection
replay start size	50000	Wait with training until the replay memory contains at least this amount of frames

The rest of the parameters not present here are as supplied by Table A.1. The combination in this table is gotten from Mnih et al., 2015, which tended to work well for some games and considerably sped up training because less updates are performed.
It is however sufficient for comparisons.

Table A.3: *LSTM* basic parameters

Hyperparameter	Value	Description
nonlinearity	tanh	Nonlinearity applied to output
gradient clipping	10	Gradients are clipped to 10

All gates are constructed with all weights initialized randomly according to a normal distribution with standard deviation 0.1. They all apply a sigmoid nonlinearity to their output.
The cell state unit has its weights initialized likewise except for the cell state value which gets initialized to 0. A tanh nonlinearity is applied to the output of this unit.

	(2,2)	(2,2) RGB	DQN-4
(2,2)	1	0.05	0.05
(2,2) RGB	0.05	1	0.05
DQN-4	0.05	0.05	1

Table A.4: Wilcoxon signed-rank test p-values for the time to a reward of 10 for the 3D convolutional architectures on the game Pong.

	LF shared	LF separate	LF shared (RGB)	DQN-4
LF shared	1	0.275	0.05	0.05
LF separate	0.275	1	0.05	0.05
LF shared (RGB)	0.05	0.05	1	0.05
DQN-4	0.05	0.05	0.05	1

Table A.5: Wilcoxon signed-rank test p-values for the time to a reward of 10 for the Late Fusion architectures on the game Pong.

	3D Conv	LF	3D Conv (RGB)	LF (RGB)	DQN-4
3D Conv	1	0.05	0.05	0.05	0.05
LF	0.05	1	0.05	0.05	0.05
3D Conv (RGB)	0.05	0.05	1	0.513	0.05
LF (RGB)	0.05	0.05	0.513	1	0.05
DQN-4	0.05	0.05	0.05	0.05	1

Table A.6: Wilcoxon signed-rank test p-values for the time to a reward of 10 for the best instances of each architecture on the game Pong.
The LSTM architecture is left out because it never managed positive average scores.

Architecture	Steps to 10 reward
3D Conv	$1.6 \times 10^6 \pm 6.0 \times 10^4$
Late Fusion (RGB)	$1.1 \times 10^6 \pm 7.3 \times 10^4$
3D Conv (RGB)	$1.0 \times 10^6 \pm 8.3 \times 10^4$
Late Fusion	$2.6 \times 10^6 \pm 2.0 \times 10^5$
DQN-4	$2.0 \times 10^6 \pm 2.9 \times 10^4$

Table A.7: Time to threshold of an accumulated reward of 10 for Pong on the best instantiations of each architecture, excluding LSTM.

Each experiment is run 3 times except the basic DQN experiments which are all run 5 times (no matter whether 1 or 4 frames as input). An experiment is run over 100 epochs of each 50000 steps. After each epoch a test phase takes place to calculate gathered rewards over 10000 steps. These are then averaged and represent a single point. The average Q-values are calculated over 3200 states that are randomly gathered at the start of the experiment.
Where rewards or average Q-values are illustrated in a graph, the graph is smoothed with a weighted running average with a window of 5 to increase visibility.
Where the Wilcoxon signed-rank test is applied, it tests the null hypothesis that the samples are drawn from the same distribution.

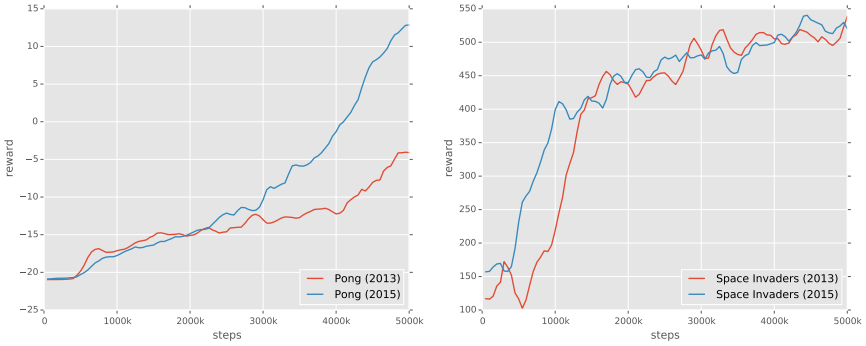
Table A.8: Basic experiment setup

APPENDIX

B

SUPPLEMENTARY EXPERIMENTS

Figure B.1: Comparison of basic DQN network performance



A comparison of learning curves for the two architectures described in Mnih et al., 2013 and Mnih et al., 2015 with network structure depicted in Figure 6.1 and parameters in Table A.2.
Some games benefit more from increased network capacity than others.

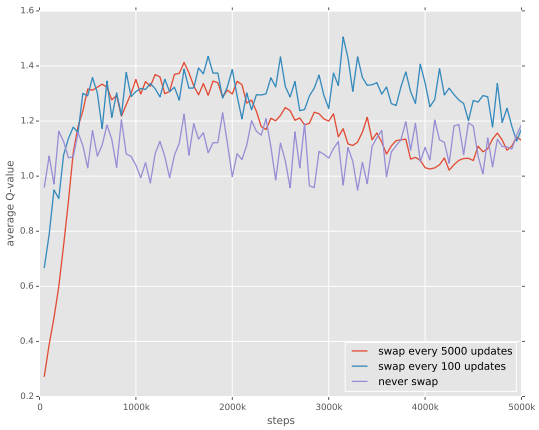


Figure B.2: This graph depicts three runs of the game Space Invaders, each with a different interval for swapping the target Q-network, a measure that has been added to stabilize learning.
The run that uses no separate target network does not really seem to learn any Q-values reliably while the network with the longest stable interval performs least noisily and actually seems to learn values.

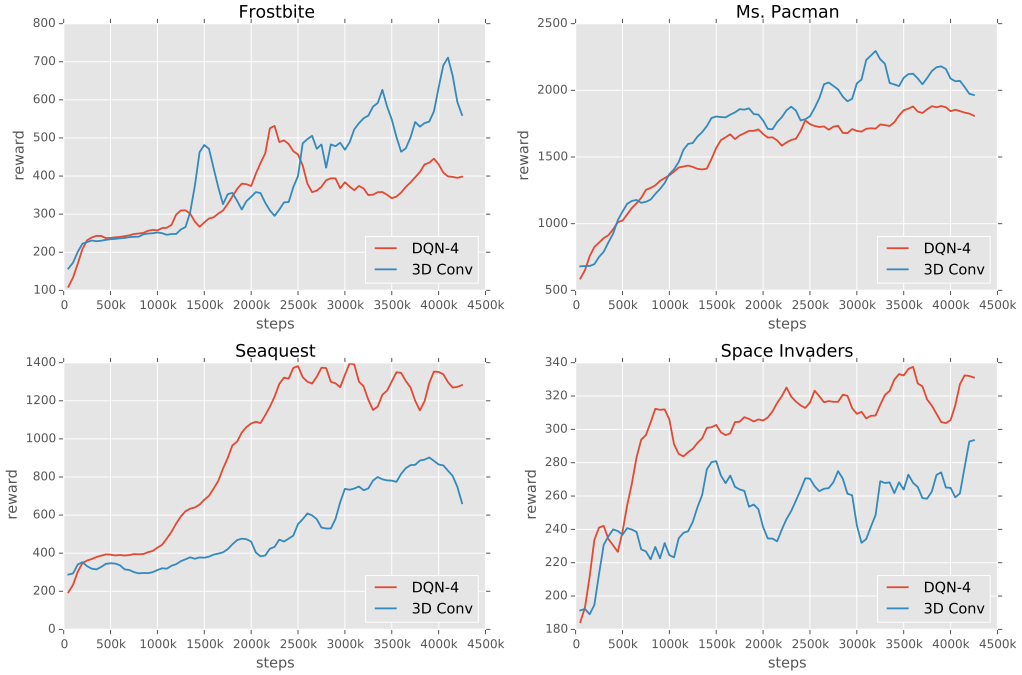


Figure B.3: Comparison of rewards over time of the DQN network versus a linear learner on 4 different games. Note that the linear results are not averaged, these are just from a single run.
The linear model exists out of a single fully-connected layer.

BIBLIOGRAPHY

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

Bakker, B. (2001). Reinforcement Learning with Long Short-Term Memory. *Nips 2001*.

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, *47*, 253–279.

Bialek, W., Rieke, F., De Ruyter Van Steveninck, R. R., & Warland, D. (1991). Reading a neural code. *Science*, *252*(5014), 1854–1857.

Cassandra, A. R., Kaelbling, L. P., & Littman, M. L. L. B. .-.-. C. (1994). Acting optimaly in partially observable stochastic domains. *Aaai*, 1023–1023.

Cottreil, G. W. (1991). Extracting Features From Faces Using Compesssion Networks: Face, Identity, Emotion, and Gender Recognition Using Holons. *Proceedings of the 1990 Summer School on Connectionist Models*, 328–338.

Dahl, G. E., Yu, D., Deng, L., & Acero, A. (2012). Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, *20*(1), 30–42.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, *12*, 2121–2159.

Friedman, J., Hastie, T., & Tibshirani, R. (2001). *The elements of statistical learning*. Springer series in statistics Springer, Berlin.

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, *36*(4), 193–202.

Google. (n.d.). Google self-driving car project. Retrieved August 20, 2016, from https://www.google.com/selfdrivingcar/

Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv preprint*. arXiv: 1308.0850v5

Graves, A., Mohamed, A.-r., & Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *2013 ieee international conference on acoustics, speech and signal processing* (pp. 6645–6649). IEEE.

Hausknecht, M. & Stone, P. (2015). Deep Recurrent Q-Learning for Partially Observable MDPs. *arXiv preprint*. arXiv: 1507.06527

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv preprint*. arXiv: 1502.01852

Hinton, G., Srivastava, N., & Swersky, K. (2012). Neural networks for machine learning. *Coursera, video lectures*.

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. *Diploma, Technische Universität München*, 91.

Hochreiter, S. & Urgen Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735–1780.

Ji, S., Yang, M., Yu, K., & Xu, W. (2013). 3D convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence*, *35*(1), 221–31.

Kaelbling, L. P., Littman, M. L., Moore, A. W., & Hall, S. (1996). Reinforcement Learning: A Survey. *Journal of Artiicial Intelligence Research Submitted*, *4*(9995), 237–285.

Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., & Fei-Fei, L. (2014). Large-Scale Video Classification with Convolutional Neural Networks. *2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 1725–1732.

Kohavi, R. et al. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai* (Vol. 14, *2*, pp. 1137–1145).

Lawrence, S., Giles, C. L., Tsoi, A. C., & Back, A. D. (1997). Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, *8*(1), 98–113.

Lecun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.

LeCun, Y. [Y.], Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation Applied to Handwritten Zip Code Recognition.

LeCun, Y. [Yann], Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2323.

Leung, M. K. K., Xiong, H. Y., Lee, L. J., & Frey, B. J. (2014). Deep learning of the tissue-regulated splicing code. *Bioinformatics*, *30*(12), i121–i129.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... Wierstra, D. (2016). Continuous Control With Deep Reinforcement Learning. *arXiv preprint*. arXiv: 1509.02971v2

Lin, L. & Mitchell, T. (1992). Memory approaches to reinforcement learning in non-Markovian domains. *Artificial Intelligence*, *8*(7597), 28.

Lin, L.-J. (1993). *Reinforcement learning for robots using neural networks*. DTIC Document.

Lovejoy, W. S. (1991). A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research*, *28*(1), 47–65.

McCallum, R. A. (1995). Instance-based utile distinctions for reinforcement learning with hidden state. In *Icml* (pp. 387–395).

Michie, D. & Chambers, R. A. (1968). BOXES: An Experiment in Adaptive Control. *Machine Intelligence 2*, 137–152.

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

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. *NIPS*, 1–9.

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.

Moore, A. W. & Atkeson, C. G. (1993). Prioritized Sweeping: Reinforcement Learning with Less Data and Less Real Time. *Machine Learning*, 103–130.

Nair, V. & Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. *Proceedings of the 27th International Conference on Machine Learning*, (3), 807–814.

Pascanu, R., Mikolov, T., & Bengio, Y. (2012). On the difficulty of training recurrent neural networks. *Proceedings of The 30th International Conference on Machine Learning*, (2), 1310–1318.

Paugam-Moisy, H. & Bohte, S. (2012). Computing with spiking neuron networks. In *Handbook of natural computing* (pp. 335–376). Springer.

Pavlov, I. P. (1927). *Conditional reflexes: An investigation of the physiological activity of the cerebral cortex*. H. Milford.

Ring, M. B. (1994). *Continual Learning In Reinforcement Environments* (Doctoral dissertation, University of Texas at Austin).

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

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, *5*(3), 1.

Rummery, G. A. & Niranjan, M. (1994). *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering.

Sammut, C. & Webb, G. I. (2011). *Encyclopedia of machine learning*. Springer Science & Business Media.

Schaul, T., Quan, J., Antonoglou, I., Silver, D., & Deepmind, G. (2016). Prioritized Experience Replay. *arXiv preprint*. arXiv: 1511.05952v4

Springenberg, J. T., Dosovitskiy, A., Brox, T., & Riedmiller, M. (2015). Striving for Simplicity: The All Convolutional Net. *Iclr*, 1–14.

Sutton, R. S. & Barto, A. G. (1998). *Reinforcement learning: an introduction*. MIT press.

Thorndike, E. L. (1911). Animal Intelligence.

Turing, A. M. (1948). Intelligent machinery, a heretical theory. *The Turing Test: Verbal Behavior as the Hallmark of Intelligence*, *105*.

Watkins, C. J. C. H. [Christopher J C H] & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3-4), 279–292.

Watkins, C. J. C. H. [Christopher John Cornish Hellaby]. (1989). *Learning from delayed rewards* (Doctoral dissertation, University of Cambridge England).

Wolf, L. (2014). Deepface: Closing the gap to human-level performance in face verification. In *Computer vision and pattern recognition (cvpr), 2014 ieee conference on. ieee*.

Woodworth, R. S. & Schlosberg, H. (1938). Experimental Psychology Holt. *New York*, 327–328.

Xiong, H. Y., Alipanahi, B., Lee, L. J., Bretschneider, H., Merico, D., Yuen, R. K. C., ... Hughes, T. R., et al. (2015). The human splicing code reveals new insights into the genetic determinants of disease. *Science*, *347*(6218), 1254806.

Zhang, Y. & Wu, L. (2011, May). Crop classification by forward neural network with adaptive chaotic particle swarm optimization. *Sensors*, *11*(5), 4721–4743.