MŰEGYETEM 1782

Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Méréstechnika és Információs Rendszerek Tanszék

Hálóarchitektúrák a mély megerősítéses tanulásban

Tudományos Diákköri Konferencia

2017. ősz

Készítette: Frendl Péter

Konzulens: Dr. Pataki Béla

Összefoglaló

A megerősítéses tanulás a gépi tanulás egy ága. Célja szekvenciális döntési problémák megoldása potenciálisan ismeretlen dinamikájú környezetekben. A feladat olyan stratégia meghatározása, amelyet felhasználva egy ágens maximalizálni tud egy valamilyen módon kumulált jutalmat az ismeretlen környezetben. A probléma az általánossága miatt sok egyéb tudományágban, például a közgazdaságtanban, pszichológiában és az idegtudományban is kutatott.

Idáig a megerősítéses tanulás alkalmazása számítási-és memóriakorlátok miatt kis állapot-és cselekvésterű környezetekre korlátozódott. A közelmúltban a hardver technológia és a mélytanulás (deep learning) fejlődése miatt lehetővé vált ezen módszerek komplex környezetekben való alkalmazása. A mély megerősítéses tanulás lehetővé tette a világ legjobb játékosainak legyőzését a Go játékban, a vizuális megfigyelésekből való tanulást és a komplex háromdimenziós helyzetváltoztatási feladatok megoldását.

Az eddigi kutatómunka a területen elsősorban a tanítási módszerek és az algoritmusok fejlesztésével foglalkozott. Bár a mélytanulás egyik legeredményesebb alkalmazási területe a képfelismerés, és számos vizuális megfigyeléseket használó megoldás létezik, kevés az olyan eredmény, amely az ágensmodellek képfeldolgozó komponenseivel foglalkozik. A képfelismerésben használt mély architektúrák és a regularizációs módszerek az eddigi próbálkozások szerint rosszul teljesítenek a megerősítéses tanulási feladatokban, így a teljesítmény javítására irányuló strukturális módosítások kutatása háttérbe szorult.

Ebben a dolgozatban a mély neurális háló ágens modellek strukturális változtatásának az ágens teljesítményére gyakorolt hatását vizsgálom a megerősítéses tanulás témakörben. A szükséges irodalmi háttér bemutatása után megmutatom, hogy hogyan teljesít az advantage actor critic módszer különféle környezetekben az egyik legelterjedtebb ágens modellt használva. Ez után elemzem, hogy a modellen alkalmazott különböző változtatások milyen hatást gyakorolnak az ágens teljesítményére.

Abstract

Reinforcement learning is a branch of machine learning. It is a general framework devoted to solving sequential decision problems in potentially unknown environments by finding the optimal way to make decisions. It is concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. The problem, due to its generality, is studied in many other disciplines, such as economics, psychology and neuroscience.

Historically, applications of reinforcement learning have been confined to environments with a small number of well-defined states and actions due to memory and computational constraints. Recent advances in hardware technology and deep learning have made it possible to apply these methods to highly complex environments as well. Deep reinforcement learning has been used to beat the best players in the world in the game of Go, to learn from visual input, and to solve three-dimensional locomotion tasks.

Research in the field has mainly focused on improving the training and optimization methods. Although one of the most significant applications of deep learning is image recognition, there has been little research published that analyzes the image processing part of the agent models that operate on visual input. Deep architectures and regularization methods used in the computer vision literature tend to perform poorly in the reinforcement learning setting, and structural modifications as a way of improving performance have been mostly sidelined.

In this paper, I study how altering the model structure of a deep neural network agent that observes its environment visually affects its performance in the reinforcement learning setting. After introducing the background literature, I show how the advantage actor critic reinforcement learning method performs in multiple visually observable environments using one of the most commonly used agent models. I then analyze how changes made to this model alter the agent’s performance.

Table of Contents

1. Introduction

Reinforcement learning has been gaining traction in recent years by the advancement of deep learning technologies. By combining reinforcement learning algorithms with deep neural networks, it has become possible to train software agents to learn to perform complex tasks from high-dimensional raw sensory feedback.

This possibility has enabled new applications of artificial intelligence technology, since being able to learn useful behaviours from rich, raw sensory data enables artificial intelligence agents to operate without a dependence on features extracted from the environment using algorithms implemented by humans. In essence, current state of the art deep reinforcement learning methods are the closest we have to general artificial intelligence. Other areas of deep learning are also flourishing: new state-of-the-art results are published every few weeks in computer vision, machine translation, speech recognition and other disciplines. Because of this, an unprecedented amount of research and investments are being applied to reinforcement learning and deep learning in general.

In the reinforcement learning literature, the main focus of the research done so far has been finding algorithms that that can optimize the agent models well for various types of environments. However, it is well known from the application of deep learning to computer vision that the performance of a neural network depends a lot on the internal structure of the network. Although deep reinforcement learning methods use neural networks to model the reinforcement learning agents, published research is scarcely available that addresses how the internal structure of an agent network affects performance.

In this work, I introduce two novel agent model architectures and I show through experiments that these can significantly outperform an architecture widely used by other researchers in the Atari [1] domain, showing the importance of developing efficient neural network architectures for deep reinforcement learning.

2. Related Work

Combining neural network-based methods with reinforcement learning has opened new possibilities in machine learning. Mnih et al. (2013) [2] created an algorithm that can learn to play atari games from raw video data output by the Atari 2600 console. They later published an improved version of this algorithm in Nature (Mnih et al., 2015) [3]. Both of these works used simple convolutional neural networks to process the observed video stream. Most researchers since then have used the network architecture presented in the latter work, focusing on improving the training algorithm.

Most of the architectural variation in the deep reinforcement learning literature addressing the Arcade Learning Environment (ALE) [1] appears in the network outputs, resulting from the algorithms used.

The deep Q-network (DQN) method [3] approximates the action-value function, and uses this approximation to follow an ε-greedy policy. Dueling network architectures [4] split the DQN action-value approximation into a state value approximation and an action advantage approximation, enabling these models to learn state values without having to learn the effect of each action for each state. The distributional DQN [5] represents the action-value function as a discrete probability distribution, providing a rich set of auxiliary predictions and a more stable learning target in the face of stochastic environment dynamics or perceived stochasticity.

The asynchronous advantage actor-critic (A3C) method [6] approximates the state-value function and outputs a probability distribution over the available functions. The state-value approximation is used to identify when an action performed from a particular state is more useful than expected, and update the agent's policy accordingly. Jaderberg et al. [7] augments the A3C method with auxiliary tasks, adding extra outputs to the neural network. These extra tasks help the formation of useful features, improving learning speed and final performance.

Kulkarni et al. [8] approximates the successor representation (SR) [9] with a neural network, increasing the sensitivity of their model to distal changes in the reward function.

These important contributions aim at improving performance by modifying the underlying algorithms and making models produce efficient representations, while using the same basic network architecture. Fixing the model architecture is important while testing algorithms, since this makes comparison possible. It is also worthwhile to know which network architectures work well (or poorly) in the domain however, since there might be much room for improvement in this area.

There have been no studies published that examine how changes to the agent model affect agent performance in the ALE, however, and this is the focus of the present work. I fix the algorithm responsible for optimizing the agent model and study how changing the model architecture impacts performance. I show that altering the image processing part of the agent model can have significant effects on learning speed and final performance.

3. Deep Learning Background

Artificial neural networks are computational models for function approximation inspired by biological neural networks found in nature. The idea has been re-popularized in recent years, going by the name “deep learning”, thanks to achieving new state of the art results in computer vision, image segmentation, speech recognition, translation and other disciplines. This chapter contains a basic overview of the deep learning concepts relevant to this work.

3.1. Deep Neural Networks

Neural networks are computational graphs which operate on tensors, i.e. n-dimensional arrays of numerical values. They are parameterized functions that map tensors to tensors through mainly differentiable operations. The successive tensor operations are usually referred to as layers. Through these layers of computation, the neural network extracts information from the input data that is relevant to producing the desired outputs. We call the extracted information features.

In the simplest case, neural networks consist of so-called fully connected linear layers and element-vise nonlinear functions, also called activation functions, following one another. The composition of a linear and a nonlinear layer is also often referred to as a layer, since it is a basic building block of neural networks. The function implemented by such a layer is as follows:

$$y = \varphi(Wx + b)$$

where $W \in \mathbb{R}^{M \times N}$ is the weight matrix that defines the coefficients for M linear combinations of the input vector, $x \in \mathbb{R}^N$ is the input vector to the layer, $b \in \mathbb{R}^N$ is a bias vector for offsetting the sum and $\varphi(\cdot)$ denotes an element-wise nonlinear transformation. $y \in \mathbb{R}^M$ is the output vector of the layer. The above equation without the application of a nonlinearity defines a linear layer output. Therefore, a linear layer computes linear functions of the elements of its input vector. The activation breaks linearity. Without this, a succession of linear layers could be reduced to just a single one implementing the same linear function.

The process of iteratively modifying the network parameter values to produce the desired outputs is referred to as training the neural network. We can also say that the network "learns" the desired function. The goal during training is to find an input-output mapping that minimizes or maximizes a particular function of the network output. This function is called the objective function. A maximization task can be reformulated as the minimization of the negation of the objective, so I will only consider the minimization case. The objective function is also often referred to as the loss or error function.

Training the network is usually done by *gradient descent*: The loss is measured on the training data, and the gradient is calculated on the loss by taking the partial derivative of the loss with respect to each parameter. Then, the model parameters are moved in the opposite direction:

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta) ,$$

where θ represents the model parameters, $\alpha \in \mathbb{R}^+$ is the learning rate coefficient and $L(\theta)$ is the loss of the model with parameters θ.

On big datasets, the gradient is usually computed from a subset of the training data to save computational resources. We call this method *stochastic gradient descent* (SGD), since it estimates the gradient on the loss with a subset of the training data. Since deep neural networks are most often trained by SGD, They should consist of differentiable operations when possible.

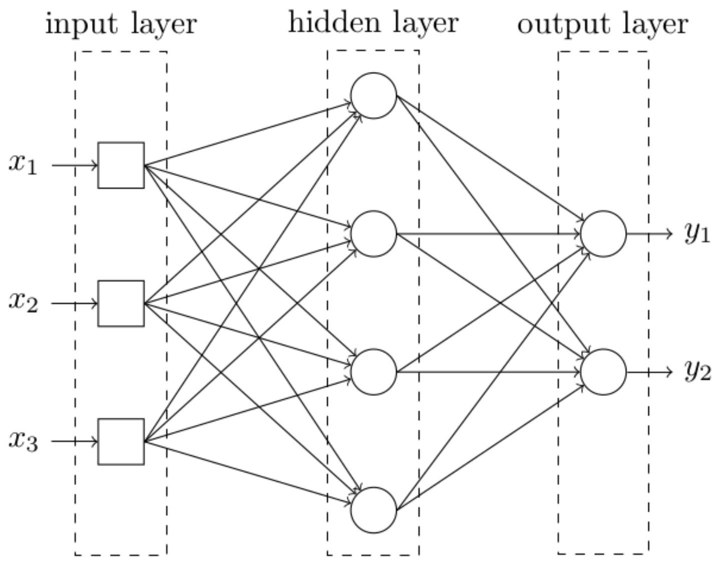


Figure 1: A simple neural network. The lines between the nodes represent linear layer weights. The data is supplied to the network through the input layer. The nodes in the hidden and output layers sum their weighted inputs and apply nonlinear functions on these sums.

We refer to the number of outputs a layer produces as the *width* or size of the layer, and we call the function that produces zero loss on the training dataset the *target function*. The universal approximation theorem states that a neural network consisting of a linear layer, a nonlinear activation function and another linear layer in this order is able to approximate the target function with an arbitrarily small loss under mild assumptions on the activation function, given the layer is wide enough [10]. However, it has been discovered that models that have many relatively narrow layers in succession instead of a few wide ones generalize better to data points not present in the training dataset.

By using multiple layers, networks are able to learn representations of data with multiple levels of abstraction [11]. Having multiple layers layers enables neural networks to reuse information extracted by a layer in subsequent layers. This enables these models to learn hierarchical representations, where simple features extracted by layers close to the input layer are used to construct more complicated or abstract features. Models that have a lot of successive layers are called deep neural networks. This is where the term deep learning originates from in machine learning.

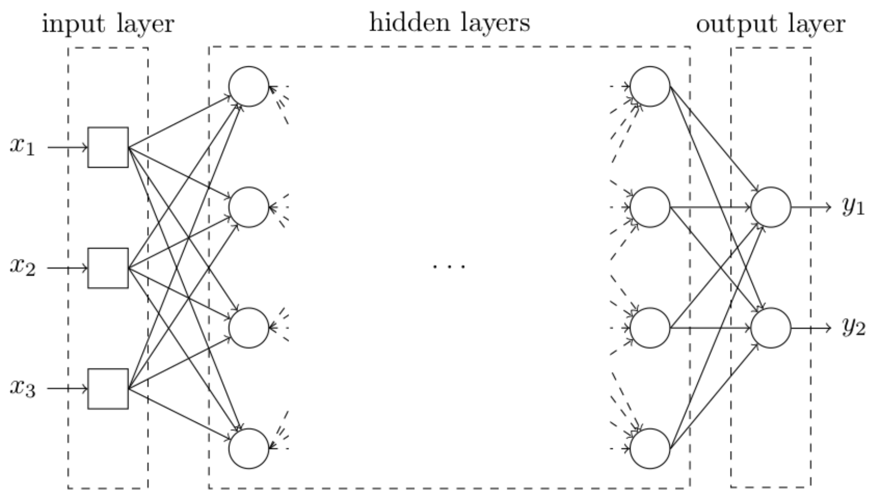


Figure 2: A deep neural network

The universal approximation theorem holds for bounded nonlinearities only, so historically, saturating functions have been used such as the sigmoid φ_s or the hyperbolic tangent function φ_h :

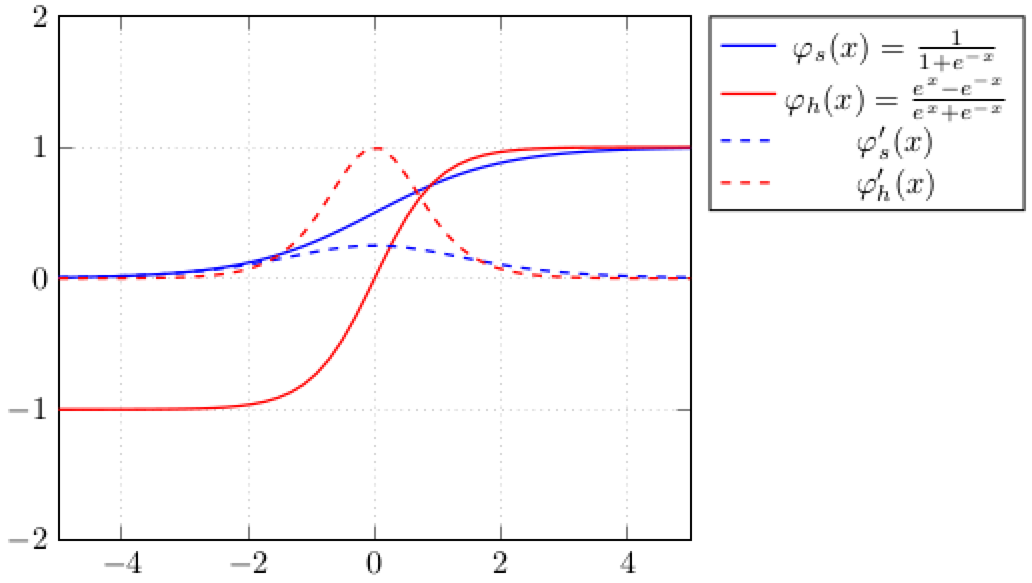


Figure 3: Sigmoid in blue, hyperbolic tangent in red

In practice, however, saturated functions don’t work well in deep neural networks due to their bounded nature. The parts of the functions where the derivatives are close to zero slow down gradient descent optimization. In addition to this, the above functions output a value with an absolute value smaller than one. Because of this, the error gradient is reduced in absolute value exponentially as it is backpropagated through neural network layers, making parameters far from the output layer hard to train. This is an issue for deep neural networks, since they have a lot of layers. In the literature, this phenomenon is called the vanishing gradient problem.

To remedy this, deep neural networks usually use a different, unbounded nonlinearity called the rectified linear unit (ReLU) φ_r :

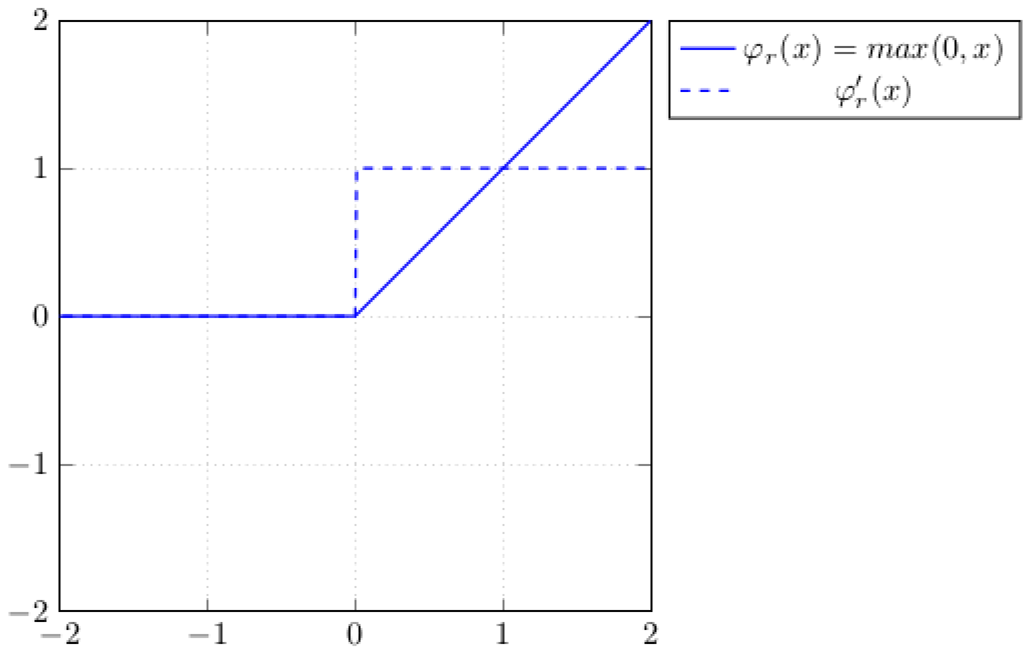


Figure 4: ReLU and its derivative

The ReLU returns its input if it is greater than zero, or zero otherwise. Since positive inputs are not downscaled, the ReLU activation facilitates easier gradient propagation, making learning faster. The function is not differentiable at $x = 0$, but in practice, the gradient at this point can be defined as $\nabla_x \varphi_r(x = 0) \equiv 0$.

3.2. Convolutional Neural Networks

Some of the most important challenges in artificial intelligence are visual tasks, since the visual processing ability of the humans is sophisticated. The most important achievements due to the deep networks' applications are connected to this field. In computer vision tasks, the goal is to extract information from image or video data. These are special data in the sense that the same local patterns can occupy different image positions. For example, a human face can appear anywhere in a photo depending on the relative position of the camera and the human, and the direction they face. We can take advantage of this characteristic of images to build better models for image recognition. The standard approach is to use convolutional neural networks (CNNs) [12].

Convolutional layers are basic building blocks of a CNN. They were invented to analyze two-dimensional data represented by uniformly sized *feature maps*. A feature map can be thought of as a view of the underlying data: it defines the value of the feature it represents for every coordinate of the data. For example, the red color channel of an RGB image describes how red the image is at any particular pixel position. Multiple feature maps can be used to describe different characteristics of the same data position. The three color channels of an RGB image work like this. Binary masks and heat maps are also feature maps, since they describe the presence or absence, or the degree of presence of a particular feature at every position of a two-dimensional geometry respectively.

A feature map is a two-dimensional tensor, or matrix of data that has width and height. If we stack d input feature maps of width w and height h along a third dimension, depth, we get a three-dimensional tensor of size (w, h, d). This how convolutional layer inputs are arranged, although the ordering of the w, h, d dimensions can vary. The individual feature maps are also called *channels*.

Convolutional layers use *kernels* to process their inputs. A kernel is also a three-dimensional tensor. It has a width and a height that are less than or equal to w and h and a depth that is equal d. The kernel is moved along the input data using some step size for every dimension, creating a linear combination of the values covered at every position using the kernel tensor values as coefficients. We also call these coefficients weights. The resulting values are stacked into a two-dimensional tensor according to the position of their source values, creating a new feature map. A convolutional layer uses multiple kernels with different coefficients to produce multiple feature maps of the input data.

The outputs of a convolutional layer are typically passed through an element-wise nonlinearity, usually ReLU. As data propagates through the CNN, the feature map resolution is incrementally reduced while the number of kernels is increased. The reduction is done by using a stride greater than one or by applying a pooling operation after the nonlinearity, typically *max-pooling*. Max-pooling works by dividing each feature map up into uniform non-overlapping rectangular areas (usually 2×2 squares). Then, a new feature map is generated from each input map by taking the maximum value from each area.

Blocks consisting of the operations described above are stacked to produce the CNN. After the spatial resolution of the data is sufficiently reduced, it is usually converted into a vector of length $w \times h \times d$ and fed into a fully connected layer, from which point the network operates in the way described in *section 3.1*. This is not necessarily true in use-cases such as generative modeling and image segmentation, but these applications are out of the scope of this work.

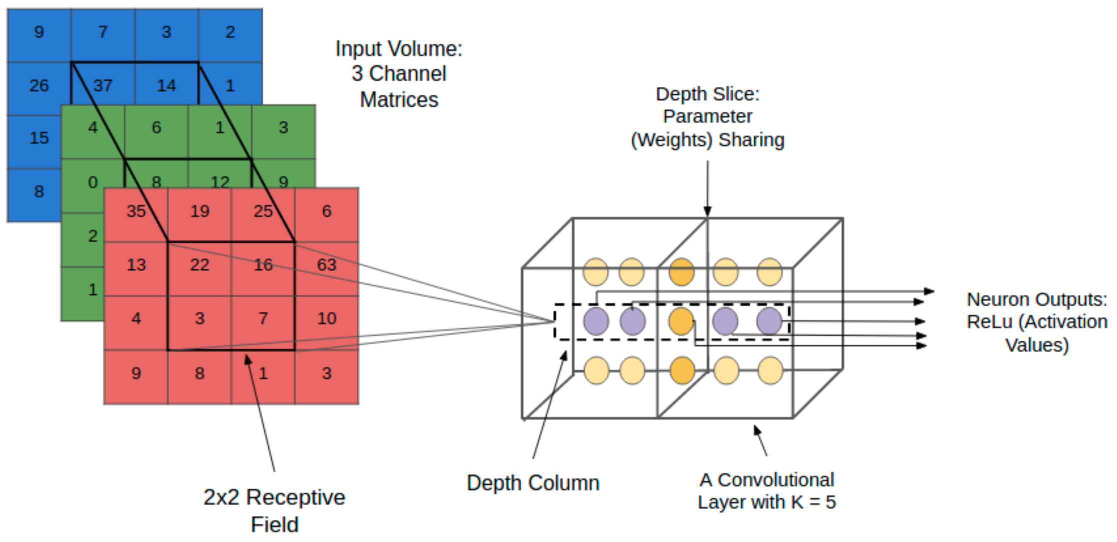

Input Volume: 3 Channel Matrices
Depth Slice: Parameter (Weights) Sharing
Neuron Outputs: ReLu (Activation Values)
2x2 Receptive Field
Depth Column
A Convolutional Layer with K = 5

Figure 5: The convolution operation

Kernels use the same set of weights in different parts of the input image, which is a form of weight sharing. Because of this, once a pattern is learnt by a kernel, it can be recognized at multiple positions in the input image. This is also referred to as translational invariance. This construction exploits regularities found in images to use network parameters more efficiently, making learning faster and resulting in better final performance.

3.3. Recurrent Neural Networks

When dealing with sequential data, we need to keep track of temporal dependencies. This necessitates the introduction of an internal state into the network. Neural networks with an internal state are called recurrent neural networks (RNNs).

A recurrent layer, in addition to its input weights and its nonlinear activation function, also has a set of recurrent weights, which are used to feed back into the layer its output produced at the previous timestep.

The most widely used RNN variant is the long short-term memory (LSTM) [13]. A single LSTM layer is governed by the following equations:

$$i_t = \varphi_s(W_i x_t + U_i h_{t-1} + b_i)$$

$$f_t = \varphi_s(W_f x_t + U_f h_{t-1} + b_f)$$

$$g_t = \varphi_h(W_g x_t + U_g h_{t-1} + b_g)$$

$$o_t = \varphi_h(W_o x_t + U_o h_{t-1} + b_o)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ g_t$$

$$h_t = \varphi_s \circ \varphi_h(c_t)$$

Variables

- x_t : input vector
- h_t : output vector
- c_t : cell state vector
- $W,\ U,\ b$: parameter matrices and vector
- f_t, i_t, o_t : gate vectors
    - f_t : Forget gate vector. Weight of remembering old information.
    - i_t : Input gate vector. Weight of acquiring new information.
    - o_t : Output gate vector. Output candidate.

Activation functions

- σ_s : sigmoid function
- σ_h : hyperbolic tangent

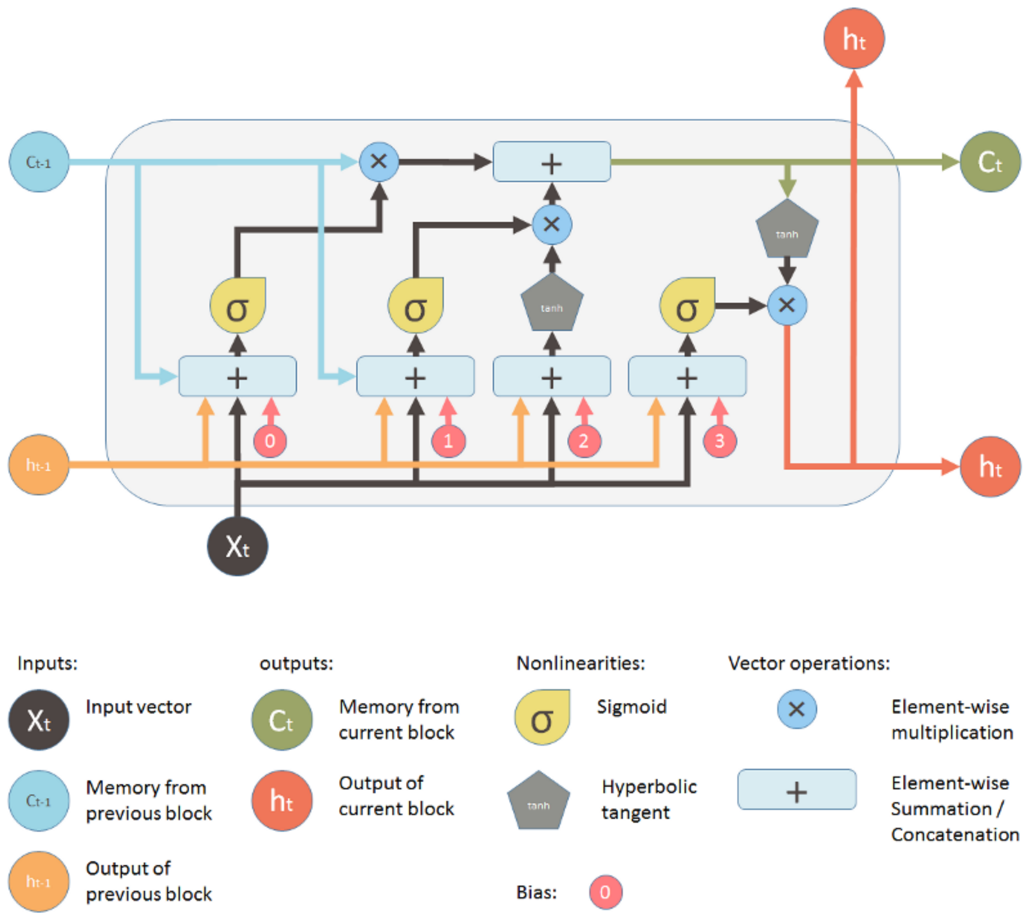


Figure 6: the internals of an LSTM

LSTM networks are adept at “remembering” long term dependencies in sequential data. This is thanks to their gating mechanisms, which are able to protect data from modification or to overwrite it when needed. Because of this protection, the networks can memorize data for many timesteps. In turn, this helps gradient information backpropagate through a large number of steps, enabling the network to learn long term dependencies.

During gradient backpropagation, LSTMs can be treated as deep neural networks with shared inter-layer connections:

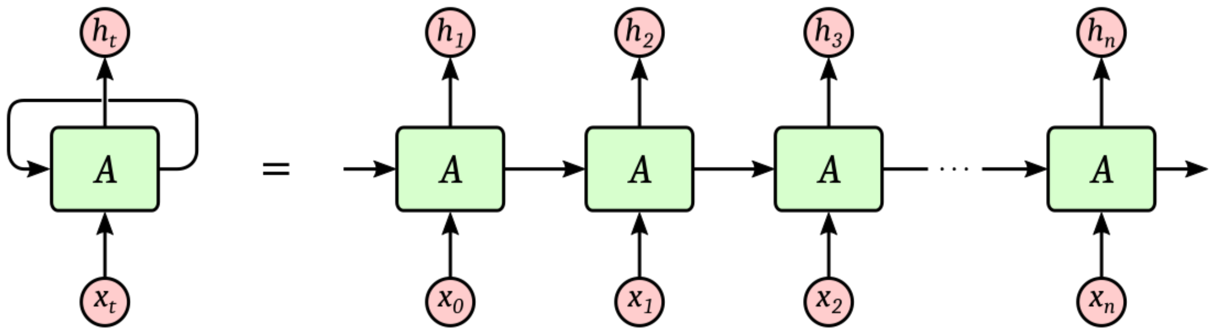


Figure 7: LSTM unrolled in time

The gradients can then be computed for the LSTM in the same manner as for a feedforward network. This method is commonly referred to as backpropagation through time (BPTT). In practice, the internal state is treated as just another input tensor to the network [14]. This makes modeling easier.

4. Reinforcement Learning Background

Reinforcement learning is a tool for solving the problem of learning from interaction with an environment to achieve some task in it. The learner is called the agent. This agent observes its environment and responds to these observations by selecting from available actions. These actions affect the observed environment, resulting in an interactive process. The environment also provides the agent with rewards (or punishments: negative rewards). Rewards are distinguished numerical values, the sum of which the agent tries to maximize over time.

In this chapter, I will introduce the reinforcement learning framework, describe some RL methods and then extend these to the deep learning setting. I have used Sutton and Barto's book *Reinforcement Learning: An Introduction* [15] as a source for most of this chapter.

4.1. Markov Decision Processes

A markov decision process, or MDP is a 6-tuple $(S, A, s_0, P, R, \gamma)$, where

- S is a set of states,
- A is a set of actions,
- $p_0(s)$ is the probability that s is the starting state,
- $p(s'|s, a)$ is the probability that performing action a in state s will lead to state s' at the next time step (transition probability),
- $r(s, a, s')$ is the (expected) value of the immediate reward received for transitioning from state s to state s' through action a,
- $\gamma \in [0, 1]$ is the discount factor, which represents the difference in importance between present and future rewards.

The above properties fully define an interactive dynamic environment. The MDP can be stochastic or deterministic. In case the reward function is stochastic, r would map to a probability distribution over the possible reward values.

The state represents whatever information is available to the agent. It contains, but does not necessarily solely consist of sensory inputs from the last timestep. It can be a preprocessed version of, or even a structure built up over time from the sequence of the agent's observations. MDPs satisfy the *Markov property*:

$$Pr\{R_{t+1}, S_{t+1} | S_0, A_0, R_1, ..., S_t, A_t\} = Pr\{R_{t+1}, S_{t+1} | S_t, A_t\}$$

where S_t is the state observed at time t, A_t is the action taken upon observing this state and R_{t+1} is the immediate reward for the transition (S_t, A_t, S_{t+1}).

This means that given the last state and the action taken from that state, all preceding states are irrelevant as far as the system dynamics are concerned.

4.2. Value Functions

A *trajectory* is a particular sequence of states, actions and rewards $\{s_0, a_0, r_1, s_1, a_1, r_2, s_2, ...\}$ sampled from the environment the agent operates in. The *return* of a trajectory is a function of the reward sequence along that trajectory. In the simplest case, it is the sum of the rewards experienced:

$$G_t = \sum_{k=1}^{\infty} R_{t+k} \,,$$

This approach suffices for *finite-horizon* settings, where every game ends after a finite number of steps. In this case, we treat the terminating state as a trapping state that returns a reward of zero indefinitely.

When dealing with *infinite-horizon* settings, i.e. settings where not every trajectory has an endpoint, the definition above could produce infinite returns, which would make comparing such trajectories difficult. There is a straightforward solution to this problem. We introduce a *discount factor* γ that determines the present value of future rewards:

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

With the above definition, a bounded reward function ensures a bounded return function: $\forall s, a, s' (|r(s, a, s')| \leq R_{max}) \Rightarrow \left|\lim_{t \to \infty} G_t\right| \leq \frac{R_{max}}{1-\gamma}$ for all trajectories. A low γ leads to a myopic return function, prioritizing short-term rewards. A high γ makes the function indifferent to the reward's delay. Assuming that the rewards are bounded, the above definition provides finite returns for even infinite trajectories.

Discounting can be viewed as a way of prioritizing rewards that are closer in time under an imperfect model of the environment, since such rewards should be easier to predict in general. Also, the higher γ is, the further back in time expected rewards affect decisionmaking for an agent that is optimized to maximize the expected return.

The state-value function describes how good it is for the agent to be in a particular state. Specifically, this function is a mapping from each state to the expected return from that state, following some policy π, where $\pi(a|s)$ is the probability that an agent following policy π takes action a from state s. The state-value function takes the following form:

$$v_\pi(s) = E_\pi\left[G_t | S_t = s\right] = E_\pi\left[\sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k} | S_t = s\right]$$

The action-value function defines the expected return given that action a is performed from state s and policy π is followed afterwards:

$$q_\pi(s,a) = E_\pi\left[G_t | S_t = s, A_t = a\right] = E_\pi\left[\sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k} | S_t = s, A_t = a\right]$$

v_π and q_π can be rewritten into recursive forms. These are referred to as the Bellman equality equations:

$$\begin{aligned}
&v_\pi(s) = E_\pi\left[G_t | S_t = s\right] \\
&= E_\pi\left[\sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k} | S_t = s\right] \\
&= E_\pi\left[R_{t+1} + \gamma \sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k+1} | S_t = s\right] \\
&= \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) \left(r(s,a,s') + \gamma E_\pi\left[\sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k+1} | S_{t+1} = s'\right]\right) \\
&= \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) \left(r(s,a,s') + \gamma v_\pi(s')\right)
\end{aligned}$$

$$\begin{aligned}
&q_\pi(s,a) = E_\pi\left[G_t | S_t = s, A_t = a\right] \\
&= E_\pi\left[\sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k} | S_t = s, A_t = a\right] \\
&= E_\pi\left[R_{t+1} + \gamma \sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k+1} | S_t = s, A_t = a\right] \\
&= \sum_{s'} p(s'|s,a) \left(r(s,a,s') + \gamma E_\pi\left[\sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k+1} | S_{t+1} = s'\right]\right) \\
&= \sum_{s'} p(s'|s,a) \left(r(s,a,s') + \gamma \sum_{a'} \pi(a'|s') E_\pi\left[\sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k+1} | S_{t+1} = s', A_{t+1} = a'\right]\right)
\end{aligned}$$

$$= \sum_{s'} p(s'|s,a) \left(r(s,a,s') + \gamma \sum_{a'} \pi(a'|s')\, q_\pi(s',a') \right)$$

v_π and q_π can also be defined in terms of each other:

$$v_\pi(s) = E_\pi \left[G_t | S_t = s \right]$$
$$= E_\pi \left[\sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k} \,|S_t = s \right]$$
$$= \sum_a \pi(a|s)\, E_\pi \left[\sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k} \,|S_t = s, A_t = a \right]$$
$$= \sum_a \pi(a|s)\, q_\pi(s,a)$$

$$q_\pi(s,a) = E_\pi \left[G_t | S_t = s, A_t = a \right]$$
$$= E_\pi \left[\sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k} \,|S_t = s, A_t = a \right]$$
$$= E_\pi \left[R_{t+1} + \gamma \sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k+1} \,|S_t = s, A_t = a \right]$$
$$= \sum_{s'} p(s'|s,a) \left(r(s,a,s') + \gamma E_\pi \left[\sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k+1} \,|S_{t+1} = s' \right] \right)$$
$$= \sum_{s'} p(s'|s,a) \left(r(s,a,s') + \gamma v_\pi(s') \right)$$

An *optimal policy* is a policy under which the expected return is maximal. The *optimal state-value function* and the *optimal action-value function* define utilities under the assumption that the agent follows an optimal policy:

$$v_*(s) = max_\pi v_\pi(s)$$

$$q_*(s,a) = max_\pi q_\pi(s,a)$$

For the action-value function, $q_*(s,a)$ gives the expected return for taking action a from state s and following the optimal policy afterwards. Therefore, q_* can be written in terms of v_*:

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

Using this formulation, we can derive the recursive form of the *Bellman optimality equation* for the state-value function. This equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$v_*(s) = max_a q_*(s,a)$$
$$= max_a E\left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\right]$$
$$= max_a \sum_{s'} p(s'|s,a)\left(r(s,a,s') + \gamma v_*(s')\right)$$

The Bellman optimality equation for $q_*(s,a)$ can be recovered from $v_*(s)$ by conditioning on a instead of maximizing over the action set. By expressing $v_*(s')$ in terms of $q_*(s,a)$, we can also make the definition recursive:

$$q_*(s,a) = E\left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\right]$$
$$= \sum_{s'} p(s'|s,a)\left(r(s,a,s') + \gamma v_*(S_{t+1})\right)$$
$$= \sum_{s'} p(s'|s,a)\left(r(s,a,s') + \gamma max_{a'} q_*(s',a')\right)$$

The value functions can be determined for specific policies. This process is called *policy evaluation*. The simplest method is to sample trajectories from the environment and define the state-value function $v(s)$ for each state s as the average of the returns encountered from s. This approximates the value $v_\pi(s) = E_\pi\left[G_t | S_t = s\right]$. The state-action value function $q(s,a)$ is defined for each state-action pair as the average of the returns encountered from state s, performing action a. This is an approximation for $q_\pi(s,a) = E_\pi\left[G_t | S_t = s, A_t = a\right]$. This approximation approach is referred to as *Monte Carlo policy evaluation*. Since the value of every state is learned independently, this method is very sample-inefficient.

Sample efficiency can be improved by taking a *dynamic programming* (DP) approach, utilizing the recursive nature of the Bellman equations. We start with some value function estimate. Then, learning is done iteratively:

$$v_{\pi_{k+1}}(s) = E_\pi\left[R_{t+1} + \gamma v_{\pi_k}(S_{t+1}) | S_t = s\right]$$
$$= \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)\left(r(s,a,s') + \gamma v_{\pi_k}(s')\right)$$

$$q_{\pi_{k+1}}(s,a) = E_\pi\left[R_{t+1} + \gamma q_{\pi_k}(S_{t+1}, A_{t+1}) | S_t = s, A_t = a\right]$$

$$= \sum_{s'} p(s'|s,a) \left(r(s,a,s') + \gamma \sum_{a'} \pi(a'|s') \, q_{\pi_k}(s',a') \right)$$

We call the above procedure *value iteration*. This method of updating state value estimates based on the value estimates of successor states is called *bootstrapping*. Since estimates are updated using other estimates, value updates can propagate between states, improving sample and update efficiency. As k approaches infinity, v_{π_k} and q_{π_k} converge to v_π and q_π respectively in the *tabular* case [15]. The function representation is considered tabular if the current estimates of the value function values are stored for every input combination separately. In practice, the value iteration is stopped when the difference between the old and the new estimates falls below a threshold value.

We can also improve upon an existing policy with the use of value functions. We can do this by first evaluating the policy and then updating it to maximize the expected return with respect to our current estimate of the value function:

$$\pi_{k+1}(s) = argmax_a \sum_{s'} p(s,a,s') \left(r(s,a,s') + \gamma v_{\pi_k}(s') \right)$$

$$\pi_{k+1}(s) = argmax_a q_{\pi_k}(s,a)$$

Alternating the policy evaluation and policy improvement steps makes the policy converge to the optimal policy π_* as the number of iterations approaches infinity [15]. This method is called policy iteration. The policy evaluation step is usually not performed until convergence, since that would be computationally expensive. Instead, a small number of iterations are performed (e.g. 1). This does not break the algorithm, π_k still converges to π_*.

The above methods assume that a model of the system dynamics (p, r) is available. Since the real dynamics are usually not available, so they can instead be approximated from the statistics of sampled trajectories of the environment. We call this method *adaptive dynamic programming* (ADP). The state transition probability $p(s'|s,a)$ can be approximated the following way:

$$\widehat{p}(s'|s,a) = \frac{N_{s,a}}{N_{s,a,s'}} ,$$

where $N_{s,a}$ is the number of actions a taken from state s and $N_{s,a,s'}$ is the number of observed transitions from s to s' after taking action a. The reward function $\hat{r}(s,a,s')$ can simply be approximated with the average value of the encountered rewards for each (s,a,s') transition.

When learning a policy using the methods described above, the algorithm can get trapped in local minima due to biases resulting from parameter initialization and the stochastic nature of the environment. To counteract this, it is necessary to introduce an exploration scheme. The simplest solution is to use an ε-greedy policy. Such a policy takes the best available action with a probability of $1-\varepsilon$, and it takes an action sampled uniformly at random from the set of available actions with probability ε, where $0 \le \varepsilon \le 1$. In the tabular case, another method for exploration is to keep count of the number of times action a was taken from state s (usually denoted as $N_{s,a}$) for each state-action pair. The value function can then be augmented with an exploration bonus that decreases as $N_{s,a}$ increases, as described by Strehl and Littman (2008) [16]. The higher this variable, the lower the utilities should be.

4.3. Temporal Difference Learning

Temporal difference (TD) learning is a method that combines the Monte Carlo and DP methods introduced in *section 4.2*. It requires less samples than the Monte Carlo method, but it does not need a model of the environment.

4.3.1. TD

Both the Monte Carlo method and the DP method use an estimate of the target to update the value function. The Monte Carlo method uses sample returns as value estimates, while the DP method uses its estimation of $v_\pi(s')$ to guide the value updates. TD learning applies both of these methods. It can be used to update the value function estimate while following a sample trajectory using the following update rule:

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

where α is the learning rate. The term $R_{t+1} + \gamma V(S_{t+1})$ is called the *TD-target,* and it is used to approximate the true value function under the current policy. The expression $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is called the *TD-error*. It is the difference of the TD-target and the current value estimate of S_t. After sampling a single reward from the return, the value estimate for state S_t bootstraps from the value estimate of state S_{t+1}. In statistical terms, bootstrapping introduces bias from the estimate $V(S_{t+1})$, but it reduces the variance introduced by full returns. The α-scaled TD-error is added to the value approximation of S_t, moving it closer to the TD-target.

4.3.2. n-step TD

More generally, returns can bootstrap after n steps. We call these *n-step returns*:

$$G_t^{(n)} = \sum_{k=1}^{n} \gamma^{k-1} R_{t+k} + \gamma^n V(S_{t+n})$$

$$V(S_t) \leftarrow V(S_t) + \alpha\left(G_t^{(n)} - V(S_t)\right)$$

With n-step TD, manipulating the bias-variance tradeoff of the TD update is possible. Returns are bias free, high variance samples of state values, while the state-value function estimate V is a stable, but biased estimate. Thus, a small n value results in low variance, high bias value function estimates, while a high n value results in low bias, high variance estimates. When $n = \infty$, the Monte Carlo method is recovered.

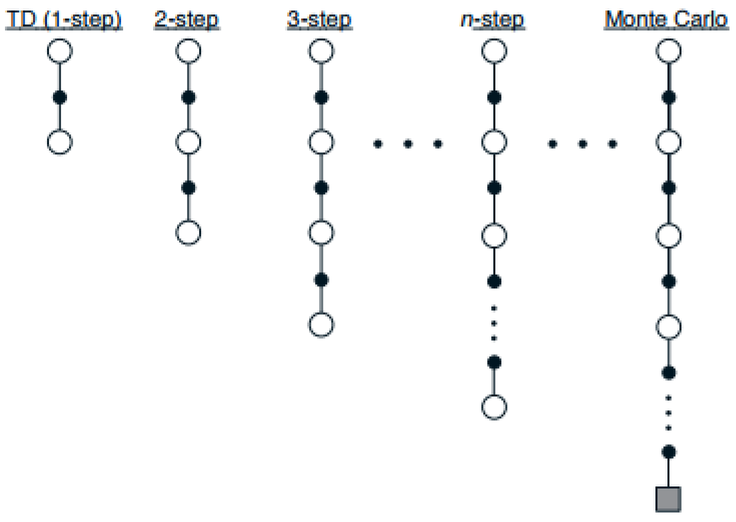


Figure 8: n-step TD [15]

4.3.3. TD(λ)

TD(λ) is a further generalization of n-step returns. This method averages over n-step returns to produce a value function estimate called the *λ-return*:

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^{(n)},$$

where $\lambda \in [0,1]$. This is a weighted average, since $(1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1} = 1, \lambda \in (0,1)$. $\lambda = 0$ gives us the one-step TD introduced in *section 4.3.1*, also known as TD(0):

$$G_t^0 = (1-0)\left(G_t^{(1)} + 0\cdot G_t^{(2)} + 0^2\cdot G_t^{(3)} + \ldots\right)$$

When a terminating state is reached, all subsequent n-step returns are equal to G_t. If we separate the post-termination terms from the main sum, we get:

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}G_t^{(n)} + \lambda^{T-t-1}G_t,$$

where T is the timestep the terminal state is reached. In this formulation, the intermediate n-step returns are weighted with their usual coefficients. The terminating state, however, is treated as a trap state, producing a reward of zero indefinitely from time T onwards. Thus, we weight this value with the remaining amount from $(1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}$, which is precisely λ^{T-t-1}:

$$(1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1} - (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1} = (1-\lambda)\sum_{n=T-t}^{\infty}\lambda^{n-1} = \lambda^{T-t-1},\ \lambda \in [0,1]$$

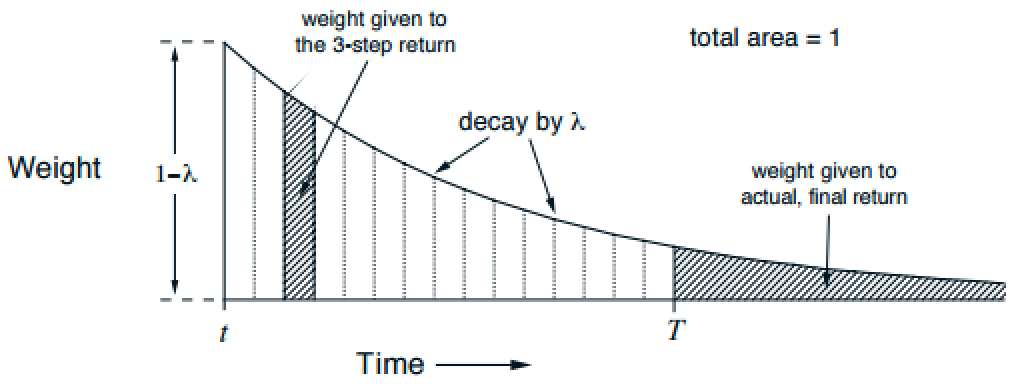

weight given to the 3-step return
total area = 1
decay by λ
Weight
1-λ
weight given to actual, final return
t
T
Time

Figure 9. TD(λ) [15]

Setting $\lambda = 1$, $G_t^1 = 0 \cdot \sum_{n=1}^{T-t-1} 1 \cdot G_t^{(n)} + 1 \cdot G_t = G_t$. In other words, the Monte Carlo method is recovered, also called TD(1).

TD(λ) can smoothly interpolate between one-step TD and the Monte Carlo method. This is the approach used by TD-Gammon [17], a program created in 1995 by Gerald Tesauro that played backgammon on expert human level.

4.3.4. TTD(λ)

Since a whole trajectory is required to calculate λ-returns, the TD(λ) method introduced in *section 4.3.3* is inapplicable in cases where we can not afford to wait until the end of a trajectory is reached. Instead, we can use *truncated λ-returns*. For time t, given data only up to a horizon h, the truncated λ-return is:

$$G_{t:h}^{\lambda} = (1 - \lambda) \sum_{n=1}^{h-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{h-t-1} G_{t:h}, \; 0 \leq t < h \leq T,$$

where $G_{t:t+k} = G_t^{(k)}$ and T is the timestep the terminal state is reached. TTD(λ) uses these λ-return estimates to update the value function. $G_{t:h}^{\lambda}$ can be rewritten in the following form [15]:

$$G_{t:t+k}^{\lambda} = V(S_t) + \sum_{i=t}^{t+k-1} (\gamma\lambda)^{i-t} \delta_t,$$

where $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is the TD-error between timesteps t and $t+1$. This formula can be used for efficient implementation. No updates are made on the first $k-1$ timesteps, and then $V(S_t), \ldots, V(S_{t+k-1})$ can be updated such that the computational cost of $G_{t:t+k}^{\lambda}$ does not scale with k: start by updating $V(S_{t+k-1})$ and work backwards, caching $\sum_{i=t}^{t+k-1} (\gamma\lambda)^{i-t} \delta_t$ between updates.

4.3.5. Off-policy TD

Q-learning is an *off-policy* TD *control* algorithm. Off-policy algorithms evaluate a different policy than the one being followed, while control algorithms define a policy to be followed. In Q-learning, the learned action-value function directly approximates the optimal action-value function q_*, independently of the policy being followed. The policy determines which state-action pairs are visited and updated, but all that is required for convergence in the *tabular* case is that all pairs continue to be updated [15]. The Q-learning action-value update rule is as follows:

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

where α is the learning rate.

The agent policy is typically derived from the learned action-value function. In the simplest case, the *ε-greedy* policy is used: The action $max_a Q_\theta(S_t, a)$ is chosen with probability $1-\varepsilon$ and a random action is sampled uniformly from A with probability ε, where $0 \leq \varepsilon \leq 1$. ε is usually annealed during training. This ensures that the convergence condition mentioned above is satisfied.

4.4. Function Approximation

Many problems have too big state and action spaces to be tackled with tabular methods. To overcome the limitations of this approach, we can employ function approximation. The best known function approximators today are neural networks: computational graphs that operate on tensors.

4.4.1. Deep Q Networks

A deep Q network (DQN) [3] is a neural network that approximates the optimal Q function of its environment through Q-learning. It then uses this approximation to behave optimally: from state s it selects action a such that the action-value approximate of the (s, a) tuple is maximal. DQNs are typically used in environments with discrete action spaces.

The temporal dynamics of the system are presented to the network in one of two ways. Either the last few observations are provided as an input, or the network contains RNN components. Typically, instead of receiving the action to be evaluated as an input, the network has an output for each available action. This way, all action-values can be computed for a state in a single *forward-pass* (feeding data through the network a single time).

During learning, the agent samples trajectories from the environment. The experienced state-action-reward triplets are stored in a replay memory, which is then used for *offline learning*, meaning that these stored experiences are reused later for learning in a non-interactive manner. The memory is sampled randomly for training samples, either uniformly or in some prioritized manner.

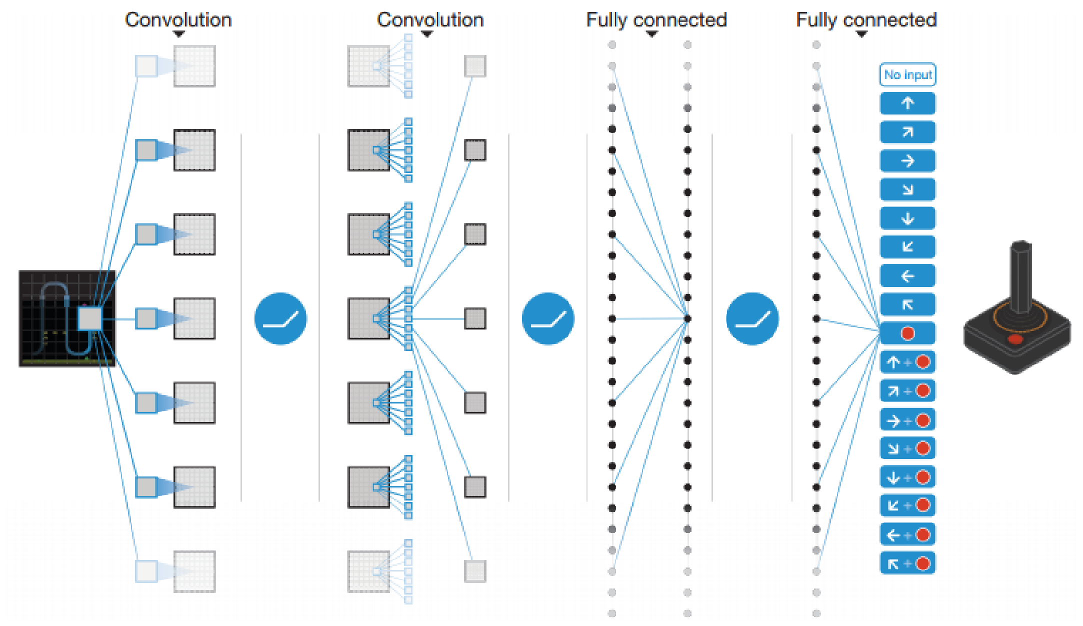


Figure 10. A DQN for playing ATARI 2600 games

A frequently used Q learning variation is the n-step Q learning, which uses n-step TD targets. For stability reasons, a snapshot θ' is created from the network parameters θ every k timesteps. These parameters are then used for calculating the n-step returns:

$$G_t^{(n)} = \sum_{k=1}^{n} \gamma^{k-1} R_{t+k} + \gamma max_a Q_{\theta'} (S_{t+n}, a)$$

This makes the TD targets constant in expectation, stabilizing learning. This is necessary because the network would otherwise overestimate the state-action values under certain conditions [18]. The agent performs Q-learning. The loss function is defined as follows:

$$E_\pi \left[\left(G_t^{(n)} - max_a Q_\theta (S_t, a) \right)^2 \right]$$

The update rule is thus the following:

$$\theta \leftarrow \theta + \alpha\nabla_\theta\left(G_t^{(n)} - max_a Q_{\theta'}(S_t, a)\right)^2,$$

Exploration has to be integrated into the agent behaviour to avoid local maxima in the policy. The ε-greedy policy is a usual choice for DQNs.

4.4.2. Deep Successor Reinforcement Learning

In *model-based planning*, the agent uses a model of the environment to simulate the consequences of future actions. Model-free methods usually store value functions for later use. In the tabular case, values are maintained separately for every input combination. Neural networks learn input-output mappings, which can also be thought of as storing function values, although they can also interpolate between known mappings. Storing values makes computing the value functions quicker, but it makes the agent inflexible to perceived changes of the MDP. With the successor representation (SR) [9], we can represent value functions in a more flexible way, while also keeping the required computation to a minimum.

The SR is defined as the expected discounted future state occupancy:

$$M(s, a, s') = E\left[\sum_{k=1}^{\infty} \gamma^{k-1} I\left[S_{t+k} = s'\right] | S_t = s, A_t = a\right],$$

where $I[\cdot] = 1$ when its argument is true and zero otherwise. As with the action-value function, we can express the SR in a recursive form:

$$M(s, a, s') = E\left[I\left[S_{t+1} = s'\right] + \gamma M(S_{t+1}, A_{t+1}, s') | S_t = s, A_t = a\right]$$

Using the SR, we can recover the action-value function the following way:

$$Q^\pi(s, a) = \sum_{s' \varepsilon S} M(s, a, s') R(s'),$$

where $R(s') = E_\pi\left[r(S_{t-1}, A_t, S_t) | S_t = s'\right]$ is the expected reward for transitioning into state s.

For large state spaces, learning the SR is intractable. Instead, we can represent each state s by a D-dimensional feature vector $\phi_s \in \mathbb{R}^D$ that is the output of a neural network. This way, we can define a feature-based SR that is the expected future occupancy of the features and denote it by $m_{s,a}$:

$$m_{s,a} = E\left[\sum_{k=1}^{\infty} \gamma^{k-1}\Theta_{t+k} | S_t = s, A_t = a\right],$$

where Θ_{t+k} is a random variable representing ϕ_s at time $t+k$. We can approximate $m_{s,a}$ using another neural network, let this be $\widehat{m}_{s,a}$. We also approximate the reward function $R(s)$ as a linear combination of the features ϕ:

$$R(s) \approx \phi_s w,$$

where $w \in \mathbb{R}^D$ is a weight vector that is learned using the squared error of the reward function approximation. Using $\widehat{m}_{s,a}$ and w, we can approximate the value function:

$$Q^\pi(s,a) \approx \widehat{m}_{s,a} w$$

When there is a change in $R(s)$, w gets modified. Since this is just a weight vector, this correction is expected to happen quickly. The reward approximation error gradients do not affect the feature representation, so $m_{s,a}$ is not affected. Thus, in case $R(s)$ changes, there is no need to relearn the SR, given a fixed policy. In principle, this means that whenever there is a change in $R(s)$, we recover a good approximation of Q^π after the quick adjustment of w. This results in an agent that can quickly adapt to changes of the reward function, while still being slow at adapting to changes of the state transition probabilities. These need not be real changes to the MDP: they can be perceived changes to the environment dynamics caused by changes in the agent's behavior.

The agent policy can be derived from the action-value approximations the same way as with the DQN (*section 4.4.1*).

Although this representation is more flexible, it does not entail significant computational overhead compared to the DQN method, making it an interesting approach. Learning a good feature representation can prove difficult, however. Tejas D. Kulkarni et al. [8] fed ϕ into a generative model that reconstructs the original input of the feature network, also using ϕ as an input to the network approximating $m_{s,a}$. They use the error gradients from these two sources to learn useful features.

4.4.3. Policy Gradient Methods

The techniques introduced in *section 4.4.1* and *section 4.4.2* used a value function estimate to pick actions. The policy can also be represented explicitly, as described below. I have used lecture 7 of *David Silver's RL course* [19] as the main source for this section.

Policy gradient methods represent the policy $\pi_\theta(a|s)$ explicitly by outputting an action distribution for every input. This distribution is then sampled to decide on the action to take. In discrete action spaces, the action distribution can be generated from some features of s by applying a linear layer with $|A|$ outputs followed by the softmax function. In the continuous case, the action distribution can be modeled by a Gaussian where the mean and standard deviation should be linear combinations of some features of s.

The softmax function is also called the normalized exponential function, and it has the following form:

$$p_j(x) = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}, \text{ for } j = 1, ..., K,$$

where x is the input vector and K is the number of elements in x.

During learning, the policy is modified to make it better. To improve upon a policy, we first have to define how to measure the quality of it. With function approximation, there are two useful ways to formulate the agent's objective [20]. The first one describes the long-term average reward achieved by the agent:

$$\rho(\pi) = \lim_{n \to \infty} \frac{1}{n} E\left[\sum_{t=1}^{\infty} R_t | \pi\right],$$

where d^π is the stationary distribution over S under policy π. The second formulation describes the expected long-term reward from a designated start state s_0:

$$\rho(\pi) = E\left[\sum_{t=1}^{\infty} \gamma^{t-1} R_t | s_0, \pi\right]$$

We can then improve the policy by following the gradient $\nabla_\theta \rho(\pi)$ upwards, modifying the model parameters in this direction. $\nabla_\theta \rho(\pi)$ is called the *policy gradient,* and following the gradient upwards is referred to as gradient ascent. According to the policy gradient theorem [20], both of the above formulations lead to the following gradient:

$$\nabla_\theta \rho(\pi_\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a), \tag{1}$$

where $d^{\pi_\theta}(s|\pi)$ is the stationary distribution over S under π_θ. In addition,

$$\nabla_\theta \pi_\theta(a|s) = \pi_\theta(a|s) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} = \pi_\theta(a|s) \nabla_\theta log \pi_\theta(a|s) \tag{2}$$

This is an expectation of $\nabla_\theta log \pi_\theta(a|s)$, which we call the score function. From *equation 1* and *equation 2,* the policy gradient is

$$\begin{aligned}\nabla_\theta \rho(\pi_\theta) &= \sum_s d^{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) \\ &= \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(a|s) \nabla_\theta log \pi_\theta(a|s) Q^{\pi_\theta}(s,a) \\ &= E_{\pi_\theta}\left[\nabla_\theta log \pi_\theta(a|s) Q^{\pi_\theta}(s,a)\right]\end{aligned}$$

We can approximate $Q^{\pi_\theta}(s,a)$ using trajectories from the environment. Using samples of the expected return $Q^{\pi_\theta}(s,a)$ to estimate the policy gradient, we get a stochastic gradient method. Updating the model parameters using G_t as an unbiased sample of $Q^{\pi_\theta}(s,a)$ is referred to as the *Monte Carlo policy gradient* method:

```
function MCPG
  Initialize θ
  for each episode $\{s_0, a_0, r_1, ..., s_{T-1}, a_{T-1}, r_T, s_T\} \sim \pi_\theta$ do
    $G \leftarrow 0$
    $\Delta\theta \leftarrow 0$
    for t = T-1 to 1 do
      $G \leftarrow r_t + \gamma G$
      $\Delta\theta \leftarrow \Delta\theta + \nabla_\theta log \pi_\theta(a_t|s_t) G$
    end for
    $\theta \leftarrow \theta + \alpha\Delta\theta$
  end for
  return θ
end function
```

Since this method uses Monte Carlo approximations of $Q^{\pi_\theta}(s,a)$, it produces very high variance stochastic gradients. We can reduce this variance using a value function approximator, as explained below.

Actor-critic methods consist of two components. The actor outputs an action distribution $\pi_\theta(\cdot|s)$ and the critic outputs a value function approximation $Q_{\theta_v}^{\pi_\theta}(s,a)$, using another set of parameters θ_v. The critic can be updated using a policy evaluation method, and we can use the approximations $Q_{\theta_v}^{\pi_\theta}(s,a) \approx Q^{\pi_\theta}(s,a)$ in the actor policy gradient update under some reasonable assumptions [20]. The action-value actor-critic method is as follows:

```
function QAC
  Initialize $\theta$, $\theta_v$
  for each episode do
    Sample $s \sim p_0(\cdot)$
    Sample $a \sim \pi_\theta(\cdot|s)$
    for each step do
      Sample transition $s' \sim p(\cdot|s,a)$, sample reward $r \sim R^{s,a,s'}$,
      Sample action $a' \sim \pi_\theta(\cdot|s')$
      $R \leftarrow stop\_grad\left(r + Q_{\theta_v}^{\pi_\theta}(s',a')\right)$
      $\Delta\theta \leftarrow \alpha\nabla_\theta log\pi_\theta(a|s)\, Q_{\theta_v}^{\pi_\theta}(s,a)$
      $\Delta\theta_v \leftarrow \beta\nabla_{\theta_v}\left(R - Q_{\theta_v}^{\pi_\theta}(s,a)\right)^2$
      $\theta \leftarrow \theta + \Delta\theta$, $\theta_v \leftarrow \theta_v + \Delta\theta_v$
      $s \leftarrow s'$, $a \leftarrow a'$
      end for
    end for
  end for
  return $\theta$, $\theta_v$
end function
```

In the above function, R is the TD target, so we do not let gradients propagate through it.

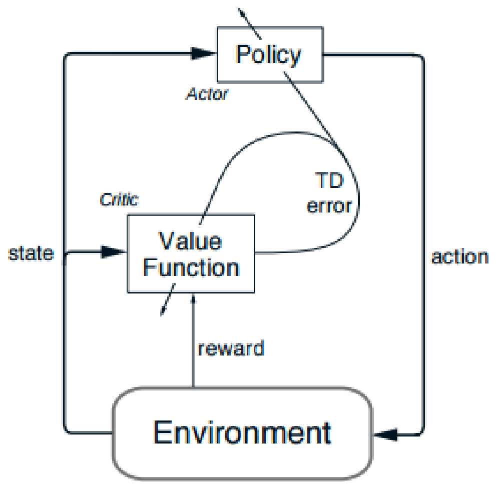


Figure 10. The actor-critic architecture

Approximating the policy gradient introduces bias into the updates. This bias can be reduced without changing the gradient's expected value by using a baseline function $B(s)$ that only depends on the state s:

$$E_{\pi_\theta}\left[\nabla_\theta log\pi_\theta(A_t|S_t)\,B(S_t)\right] = \sum_s d^{\pi_\theta}(s)\sum_a \nabla_\theta\pi_\theta(a|s)\,B(s)$$

$$= \sum_s d^{\pi_\theta}(s)\,B(s)\,\nabla_\theta\sum_a \pi_\theta(a|s)$$

$$= \sum_s d^{\pi_\theta}(s)\,B(s)\,\nabla_\theta 1$$

$$= 0$$

A good choice for the baseline is the state-value function. Subtracting the state-value function from the action-value function, we get the advantage function. This tells us how much more reward than usual we can get if we take a particular action a from state s:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$$

The policy gradient can be expressed using this function, producing lower variance gradient estimates:

$$\nabla_\theta\rho(\pi) = E_{\pi_\theta}\left[\nabla_\theta log\pi_\theta(A_t|S_t)\,Q^{\pi_\theta}(S_t,A_t)\right]$$

$$= E_{\pi_\theta}\left[\nabla_\theta log\pi_\theta(A_t|S_t)\,Q^{\pi_\theta}(S_t,A_t)\right] - E_{\pi_\theta}\left[\nabla_\theta log\pi_\theta(A_t|S_t)\,V^{\pi_\theta}(S_t)\right]$$
$$= E_{\pi_\theta}\left[\nabla_\theta log\pi_\theta(A_t|S_t)\,(Q^{\pi_\theta}(S_t,A_t) - V^{\pi_\theta}(S_t))\right]$$

$$= E_{\pi_\theta}\left[\nabla_\theta log\pi_\theta(A_t|S_t)\,A^{\pi_\theta}(S_t,A_t)\right]$$

It is therefore beneficial for the agent to work with the advantage function instead of the Q function. One possible way to do this is to estimate both $V^{\pi_\theta}(s)$ and $Q^{\pi_\theta}(s,a)$. Alternatively, we can make the following observation about the TD error of the true state-value function:

$$\delta^\pi = R_{t+1} + \gamma V^\pi(S_{t+1}) - V^\pi(S_t)$$

$$E_\pi[\delta^\pi|s,a] = E_\pi\left[R_{t+1} + \gamma V^\pi(S_{t+1})\,|S_t = s, A_t = a\right] - V^\pi(s)$$

$$= Q^\pi(s,a) - V^\pi(s)$$

$$= A^{\pi}(s,a)$$

Since δ^{π} and $A^{\pi}(s,a)$ are equal in expectation, we can use the TD error to compute the advantage policy gradient, without a need for $Q^{\pi}(s,a)$:

$$\nabla_{\theta}\rho(\pi_{\theta}) = E_{\pi_{\theta}}\left[\nabla_{\theta} log\pi_{\theta}(A_t|S_t)\,\delta^{\pi_{\theta}}\right] \tag{3}$$

The real value function is generally not available, so we use an approximate TD error instead:

$$\delta_{\theta_v}^{\pi_{\theta}} = R_{t+1} + \gamma V_{\theta_v}^{\pi_{\theta}}(S_{t+1}) - V_{\theta_v}^{\pi_{\theta}}(S_t)\,, \tag{4}$$

where $V^{\pi_{\theta}}(s)$ is a value function approximation, output by the critic. Replacing the action-value policy gradient in the QAC algorithm with the advantage policy gradient from *equation 3* using the approximate TD error from *equation 4* results in the advantage actor critic (A2C) algorithm.

The advantage function can also be estimated by averaging over n-step advantage estimates, analogously to how the TD(λ) value function estimator works. This is called generalized advantage estimation [21]. The estimate can be written in the following form:

$$\widehat{A}_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty}(\gamma\lambda)^l\left(\delta_{\theta_v}^{\pi_{\theta}}\right)_{t+l}\,,$$

where γ is the discount factor, λ controls the weighting of the n-step advantage terms and $\left(\delta_{\theta_v}^{\pi_{\theta}}\right)_{t+l}$ is the approximate TD error at time $t+l$.

To encourage exploration, we can modify the policy gradient to include the gradient of the entropy of the action distribution. This will discourage the agent from prematurely converging to suboptimal deterministic policies [6]. The new gradient becomes

$$E_{\pi_{\theta}}\left[\nabla_{\theta} log\pi_{\theta}(a|s)\widehat{A}^{GAE(\gamma,\lambda)} + \eta\nabla_{\theta}H(\pi_{\theta}(\cdot\,|s))\right]\,,$$

where η is a hyperparameter that controls the strength of the entropy regularization.

5. Test Environment

For testing, I use Atari environments from the OpenAI Gym open-source library [22]. OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms, while the Atari environments are a variety of Atari 2600 video games ran with the Arcade Learning Environment framework [1], which is in turn built on top of the Stella Atari 2600 emulator [23]. I set up the environment the same way as Schulman et al. (2017) [24] to make comparing the test results possible. I detail the setup in *section 5.2* and *section 5.3*.

5.1. Interface

Environments in the OpenAI Gym toolkit provide a uniform interface. Every timestep, the environment expects one of the available agent actions as an input, and supplies a reward value, the next observation and a “done” boolean value as outputs, signaling whether the game or simulation episode is over. Interface-wise, the Atari environments differ only in the number of actions available, and every environment has a discrete action space. Every observation is an 210×160 RGB image of the console screen. Observations are therefore tensors of the shape $(210, 160, 3)$. Alternatively, there exists a version for every Atari environment which provides the environment’s RAM states as observations. These are out of the scope of this work.

5.2. Preprocessing

To encode an observation I take the maximum value for each pixel colour value over the corresponding frame and the previous frame. This is necessary to remove flickering that is present in games where some objects appear only in even frames while other objects appear only in odd frames, an artefact caused by the limited number of sprites the Atari 2600 can display at once [3].

To make training faster, virtually all solutions apply frame-skipping in some manner. This means that we only consider a subset of the observations, we choose an action after processing every such observation and we repeat the chosen action until the next observation to be considered. Most authors use a constant frame-skip of $n_{frame-skip} = 4$, meaning that every fourth observation is considered, starting with the first one $(1, 5, 9, ...)$, and the chosen actions are repeated for the timesteps in-between, ignoring feedback from the skipped frames.

For every OpenAI Gym Atari environment, there is a version with built-in frame-skipping where $n_{frame-skip}$ is uniformly sampled from $\{2, 3, 4\}$, and another version with no frame-skipping. In this work, I follow most authors by using a frame-skip of $n_{frame-skip} = 4$. I implement this using the environment versions with no frame-skip by considering feedback only from timesteps with indices of the form $1 + in_{frame-skip}, i\varepsilon\mathbb{N}$.

The observations are also usually converted to grayscale and downscaled to make computation faster. I apply grayscale conversion and downscale the images to a 84×84 resolution. The preprocessed observations are therefore tensors of the shape (84, 84, 1), which is the input format used by Schulman et al. (2017) [24].

5.3. Stochasticity

In almost all games, the dynamics of the Stella emulator are deterministic. This means that a particular game always starts from the same position, and following a particular sequence of actions always results in the same outcome [25]. Such environments can be exploited by agents that rely on this determinism for successful performance by learning the exact outcomes of action sequences. These kinds of algorithms can be sensitive to small perturbations, and may not generalize well to stochastic environments. For this reason, many researchers have augmented the ALE environments to introduce stochasticity, using various methods. OpenAI [26] created a well-tuned implementation of the A2C algorithm that was benchmarked in Schulman et al. (2017) [24]. I also use an A2C-based agent, so I will use the same method the did: 0 to k no-op actions will be performed at the beginning of every game, selected uniformly at random. I will use $k = 30$, which is the value Schulman et al. (2017) [24] used [27]. The no-op action corresponds to the case where the console receives no input, so it is available in every game. This method is usually referred to as *no-ops starts*.

6. Algorithm

I use a synchronous version of the A3C algorithm [6] for all experiments. A3C is a policy gradient-based algorithm that runs multiple agents in different instances of the training environment in parallel processes. The agents are controlled by a shared agent model, which is updated by the processes asynchronously.

In *section 6.1*, I outline the functions of different parts of the agent model and I explain how the algorithm works. In *section 6.2*, I present the pseudocode of the training algorithm.

6.1. Outline

Each tested model has an image processing part implemented by convolutional layers, followed by a recurrent part implemented by an LSTM (*section 3.3*). The LSTM outputs a latent representation of the environment state. This is passed through a linear layer, followed by a softmax layer (*section 4.4.3*) to produce the action distribution $\pi_\theta(s)$ over the set of actions A, where θ represents the policy model parameters. The LSTM outputs are also passed through another linear layer to produce the state-value estimate $V_{\theta_v}^{\pi_\theta}(s)$ for the current environment state under policy π_θ, where θ_v represents the state-value model parameters. Therefore, most of the parameters from θ and θ_v are shared. This saves computation time, since the shared representations are only computed once during the forward pass, and the sharing of parameters encourages learning mutually beneficial representations. Since the network receives feedback from two error sources, learning such representations can become quicker, resulting in more *sample-efficient* learning, meaning that the network needs to see fewer examples to learn the target function.

Computation is distributed between a master process and $n_{workers}$ worker processes. The master is responsible for feeding observations into the neural network, providing actions to the workers, and periodically updating the agent model based on the acquired data. Each worker is responsible for executing the received actions in their own instance of the test environment and providing the resulting observations to the master. The network computations are delegated to the graphics processing unit (GPU) for faster computation time.

The master keeps track of an agent state for each worker process. Each agent instance plays in a different instance of the same environment, ran by the workers in parallel. This makes learning more stable, since the IID assumption that does not hold in the RL setting is recovered to some extent.

IID is an assumption on the training sample distribution and it stands for *Independently and Identically Distributed*. If this assumption does not hold, neural networks have a hard time learning the target task. If there are multiple stochastic agents running in different instances of the same environment, they can explore different parts of the state space at the same time, making the observations more independently distributed.

Running multiple environment instances in parallel, we can also acquire more samples in a given time period (assuming we have enough processing cores), resulting in less wall-clock training time.

The model is updated every P timesteps. Between updates, the agents interact with their environments and data is collected for the next update. Let p be the number of steps elapsed since the last model update. I use n-step returns (*section 4.3.2*) as targets for the state-value approximation of the model and I use the squared error to compute the value loss:

$$L_v(\theta_v) = E\left[\left(G_t^{(P-p)} - V_{\theta_v}^{\pi_\theta}(s_t)\right)^2\right],$$

I use a truncated version of generalized advantage estimation (*section 4.4.3*) to calculate the policy loss, also using entropy regularization (see *section 4.4.3*):

$$\widehat{A}_t^{GAE(\gamma,\lambda),P-p} = \sum_{l=0}^{P-p} (\gamma\lambda)^l \left(\delta_{\theta_v}^{\pi_\theta}\right)_{t+l}$$

$$L_\pi(\theta) = E_{\pi_\theta}\left[log\pi_\theta(a|s)\widehat{A}^{GAE(\gamma,\lambda),P-p} + \eta H(\pi_\theta(s))\right]$$

Notice the similarity between $\widehat{A}_t^{GAE(\gamma,\lambda),P-p}$ and the truncated λ-return from *section 4.3.4*. $G_{t:t+k}^{\lambda}$ gives a state-value estimate, while $\widehat{A}_t^{GAE(\gamma,\lambda),P-p}$ gives a state-advantage estimate, and their difference is is the state value (although the state values are estimated in generalized advantage estimation).

For learning representations that are mutually beneficial for the policy and value networks, it is important that the relative magnitudes of L_π and L_v are correct. For this reason, I multiply L_v by the constant β in the combined loss, which is the value that is used to compute the gradients:

$$L(\theta, \theta_v) = L_\pi(\theta) + \beta L_v(\theta_v)$$

Each training session lasts for $n_{frames-max}$ game frames in total including the skipped frames and summed over the worker processes. The learning rate α is linearly annealed over the course of learning. I use Adam [28] as the optimization method.

OpenAI [26] has created an implementation of A2C for comparison purposes as part of their baseline implementations repository [27]. There are a few differences between their implementation and mine. For advantage estimation, I use generalized advantage estimation, while they use the difference between n-step returns and the state-value function estimates. Also, I use the Adam optimization algorithm instead of RMSProp [29]. I use no gradient clipping (downscale gradients so that their L2-norm is of a threshold value in case their current L2-norm is greater than this value) and I also use a different set of hyperparameters (see *section 7.2*).

6.2. Pseudocode

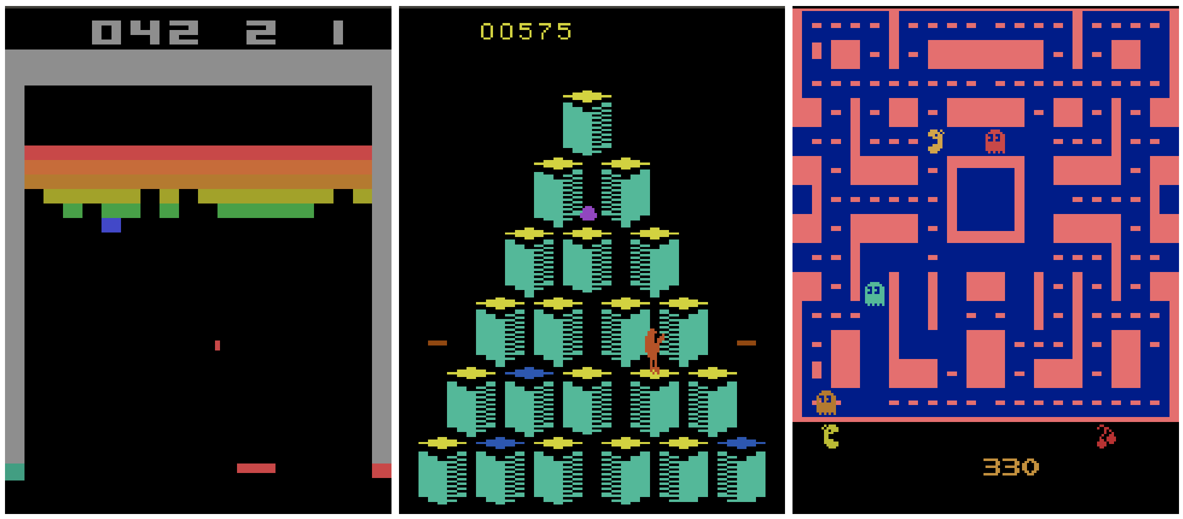
```
// Assume model parameter vectors $\theta$, $\theta_v$
Start workers $w_k : k \varepsilon \{1, 2, ..., n_{workers}\}$, let $w$ refer to all the workers ordered by their indices
Get observations $o_k \sim w_k : k \varepsilon \{1, 2, ..., n_{workers}\}$, let $o$ refer to all the observations ordered by their indices
Define batch of episode over flags $f \varepsilon \mathbb{Z}_2^K \leftarrow 1$
Create agent states $s_k, k \varepsilon \{1, 2, ..., n_{workers}\}$, let $s$ be an ordered batch of the agent states
$n_{frames} \leftarrow 0$
do
    $p \leftarrow 0$
    Create lists $T_l, T_v, T_h, T_r, T_f$
    do
        for $k \varepsilon \{1, 2, ..., n_{workers}\}$ do if $f_k = 1$, initialize agent state $s_k$ end for
        Feed $\{s, o\}$ into the model to get new $\left\{s, \pi_\theta(\cdot|s), V_{\theta_v}^{\pi_\theta}(s)\right\}$ batches
        Sample batch of actions $a \sim \pi_\theta(\cdot|s)$ and send them to $w$
        Get feedback from workers $(o, r, f) \sim w$          // $r$ is the ordered batch of rewards
        Append $log\pi_\theta(a|s)$ to $T_l$
        Append $V_{\theta_v}^{\pi_\theta}(s)$ to $T_v$
        Append $H(\pi_\theta(\cdot|s))$ to $T_h$                // $H(\cdot)$ is the information entropy
        Append $r$ to $T_r$
        Append $f$ to $T_f$
        $p \leftarrow p + 1$
    while $p < P$
    Feed $\{s, o\}$ into the model to get new $\left\{\cdot, \cdot, V_{\theta_v}^{\pi_\theta}(s)\right\}$ batches
    Append $V_{\theta_v}^{\pi_\theta}(s)$ to $T_v$
    $R \leftarrow V_{\theta_v}^{\pi_\theta}(s)$, detach $R$ from computation graph
    $\widehat{A} \leftarrow 0, L_\pi(\theta) \leftarrow 0, L_v(\theta_v) \leftarrow 0$
    for $i \in \{K, K-1, ..., 1\}$ do
        $R \leftarrow T_r[i] + (1 - T_f[i])\gamma R$
        $\delta \leftarrow T_r[i] + (1 - T_f[i])\gamma T_v[i+1] - T_v[i]$
        $\widehat{A} \leftarrow \delta + \gamma\lambda\widehat{A}$, detach $\widehat{A}$ from computation graph
        $L_\pi(\theta) \leftarrow L_\pi(\theta) - \left(T_l[i]\widehat{A} + \eta T_h[i]\right)$
        $L_v(\theta_v) \leftarrow L_v(\theta_v) + (R - T_v[i])^2$
    end for
    Update $\theta$ and $\theta_v$ with the gradients from the loss $L_\pi(\theta) + \beta L_v(\theta_v)$
    Detach $s$ from the computation graph
    $n_{frames} \leftarrow n_{frames} + n_{workers} n_{frame-skip} P$
while $n_{frames} < n_{frames-max}$
```

7. Experiments

In this section, I present my test results. I introduce the games used for the experiments in *section 7.1*. In *section 7.2*, I define the hyperparameters I used for all tested models in all games. In *section 7.2*, I introduce the examined models. I present and analyze the test results in *section 7.3*.

7.1. Environments

Most authors publish results for dozens of games ([6], [24]). Because of limited time and computational resources, I run tests on three visually and gameplay-wise quite different games: Breakout, Qbert, and MsPacman. I optimize all hyperparameters on Breakout. I use the other two games to see if the difference in performance between the tested model architectures is consistent across games. I use the same set of hyperparameters for all games.



Breakout Qbert MsPacman

Figure 12. Screenshots of the tested games

7.2. Hyperparameters

Every experiment is started with a learning rate $\alpha = 1.5 \times 10^{-4}$, annealed linearly over the course of training. I use a discount factor of $\gamma = 0.99$ and a λ of 0.9 for general advantage estimation. Entropy regularization is applied with a weight of $\eta = 0.01$. The value approximation loss is given a weight of $\beta = 0.5$ in the combined loss. I update the agent model every $P = 20$ timesteps and $n_{workers} = 4$ workers run in parallel. Each training session lasts for $n_{frames-max} = 4 \times 10^7$ frames. For Adam, I use $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$.

Hyperparameter	Value
α	1.5×10^{-4}
γ	0.99
λ	0.9
η	0.01
β	0.5
P	20
$n_{workers}$	4
$n_{frames-max}$	4×10^{7}

Table 1. Hyperparameter values used in all tests

7.3. Model Architectures

Because of computational constraints, I limit the number of architectures to test to three. One of these is based on the architecture from Mnih et al. (2015) [3] mentioned in *section 2*. I use the version from Mnih et al. (2016) that appends an LSTM with 256 outputs after the fully connected layer, since it works better [6]. This is the *baseline architecture* I compare my variations against.

The baseline architecture receives input tensors of size $(84, 84, 1)$. The first convolutional layer has 32 8×8 kernels with stride 4, followed by a layer of 64 4×4 kernels with stride 2, followed by a layer of 64 3×3 kernels with stride 1. ReLU nonlinearities are applied after all convolutions to produce the layer outputs. The output of the last convolutional layer is fed into a linear layer with 512 outputs, followed by a ReLU, followed by the LSTM. A linear layer is applied to the outputs of the LSTM, followed by a softmax layer to produce the action distribution $\pi_\theta(\cdot|s)$. Another linear layer is applied to the LSTM outputs to produce the value function approximation $V_{\theta_v}^{\pi_\theta}(s)$.

I did a series of short preliminary tests on the game Breakout to narrow down the architectures that might perform better than the baseline. I then chose two architectures with promising properties to compare against the baseline. I call these *architecture A* and *architecture B*.

Architecture A is a modification of the baseline: the fully connected layer is removed, and the kernel count of the first convolutional layer is increased from 32 to 64. In preliminary testing, this setup trained faster than the baseline.

Architecture B results from adding two extra convolutional layers with a kernel size of 5×5, a stride of 1 and kernel count of 64 before the first layer of architecture A. In preliminary testing, I observed that by adding convolutional layers before the first layer of the baseline architecture that do not change the feature map resolutions but increase the channel count, agents learn faster. Architecture B had considerably better training speed and final performance than the baseline.

Layer				baseline	A	B
Layer depth baseline/A/B	Convolutional + ReLU			Output channels		
	Kernel size	Stride	Padding			
-/-/1	[5, 5]	1	2	-	-	64
-/-/2	[5, 5]	1	2	-	-	64
1/1/3	[8, 8]	4	3	32	64	64
2/2/4	[4, 4]	2	1	64	64	64
3/3/5	[3, 3]	1	1	64	64	64
Layer depth	Fully connected			Outputs		
4/4/-	Linear + ReLU			512	-	-
5/5/6	LSTM			256	256	256
6/6/7	Linear + softmax (π_θ)			$\|A\|$	$\|A\|$	$\|A\|$
7/7/8	Linear $\left(V^{\pi_\theta}_{\theta_v}\right)$			1	1	1

Table 2: Model architectures

7.4. Test Results

I run all training sessions with the algorithm presented in *section 6*, using the hyperparameters defined in *section 7.2*. I run three experiments for every game-architecture pair.

During training, I log the scores one of the workers receives. These values tend to be high in variance, so I smooth them by averaging over the last 100 scores acquired. This is a common smoothing technique in the literature [24] [25]. Other frequent methods are to stop training and evaluate the agent performance every N (e.g. one million) frames, or to evaluate the agent only at the end of training. The approach I use lets me visualize the scores received over time and compare against the A2C implementation of Schulman et al. (2017) [24], since they use the same smoothing method. By plotting the smoothed scores received against the number of frames elapsed, the learning progress of the agent can be visualized, so these plots are referred to as *learning curves* in RL. The effects of the smoothing can be seen in *figure 13*. Raw scores are displayed for every game-architecture pair in *figure 18* in the *Appendix*.

For every test run, I calculate the final performance of the agent by averaging over the scores of the last 100 games played, as Schulman et al. (2017) [PPOA] did. I present the mean and standard deviation of these values in *table 3*, grouped by game and architecture, also including the results from Schulman et al. (2017).

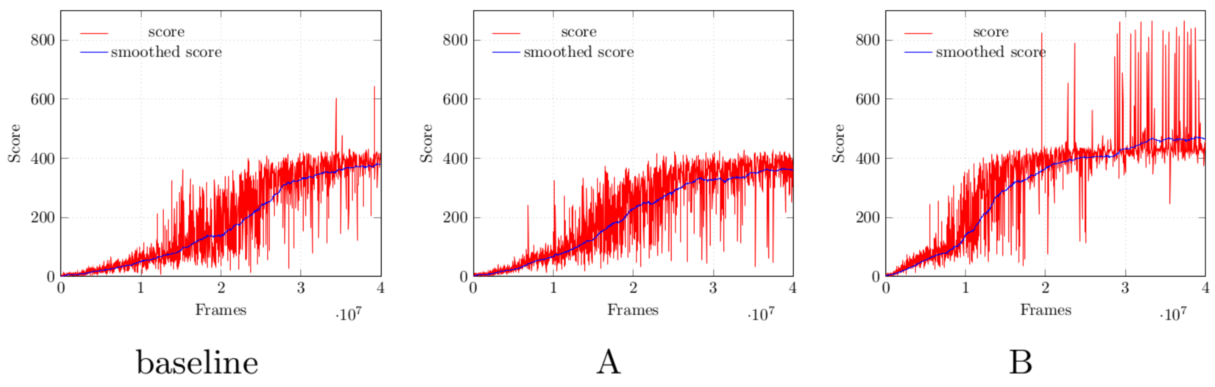


Figure 13. Raw and smoothed learning curves on the game Breakout. From left to right, the result of the first test run is displayed for the baseline, A, and B respectively.

The task being learned can have a substantial effect on the learning curve in RL. In Breakout, the agent has to clear the playing field of blocks by hitting them with a ball, getting some points for each block, 448 in total. However, the fewer blocks there are, the more difficult it is to hit one. If all the blocks are cleared, another batch of blocks appear, making scoring points easy again. We can see the effects this has on the raw scores in *figure 13*. Agents have difficulty scoring above the 448 point threshold, but when they do, they are often able to score a lot more points than on average.

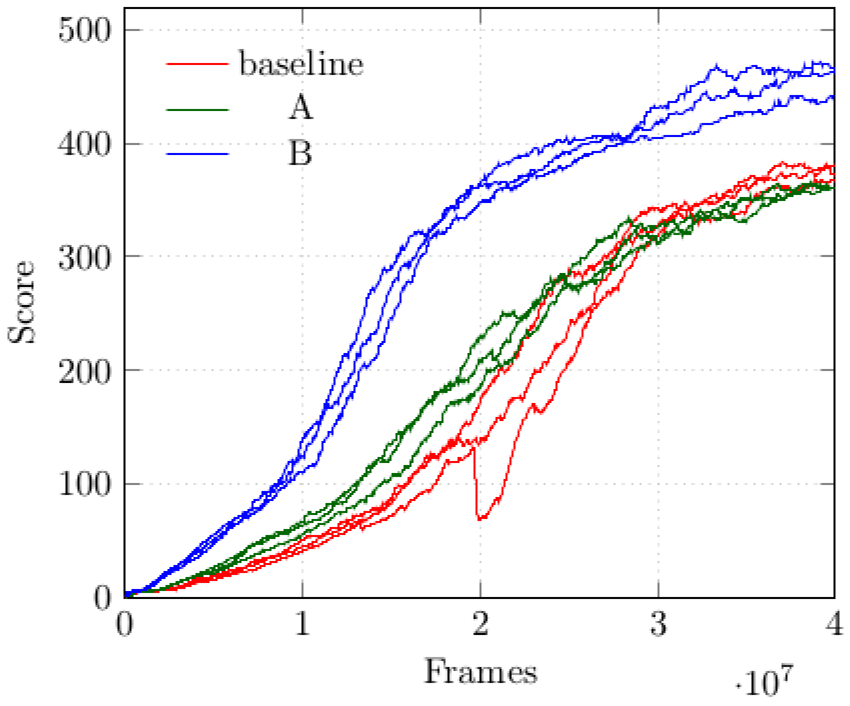


Figure 14. The learning curves of all test runs on Breakout for the the baseline, A, and B. Note the effects different architectures have on learning speed and final performance.

The results of the test runs on Breakout are plotted together in *figure 14*. There are a few different things to note in this figure. First, the rate of improvement with respect to the number of frames observed differs between architectures. The baseline model is the slowest learner, followed by A, and B is the most sample-efficient in this game. Second, although A learns faster, it falls off in the long run, suggesting that there is a tradeoff between the two architectures between learning speed and maximum performance, for this particular game at least. Third, one of the test runs of the baseline exhibits instability in the learning process. This is a common occurrence in RL when the hyperparameters are not ideal, but it can happen anyways. After the drop in performance, the agent quickly recovers. It might even seem like this agent is improving faster than the others during the catch-up phase, but the steepening of its learning curve that lasts for roughly a million frames is only a consequence of the score smoothing. This becomes apparent in *figure 15*, where I have visualized the learning curve with the raw score values. A sudden drop in performance can be observed around the 2 million frame mark that gets corrected in a few game episodes. It has a long-term effect on the smoothed score values, however.

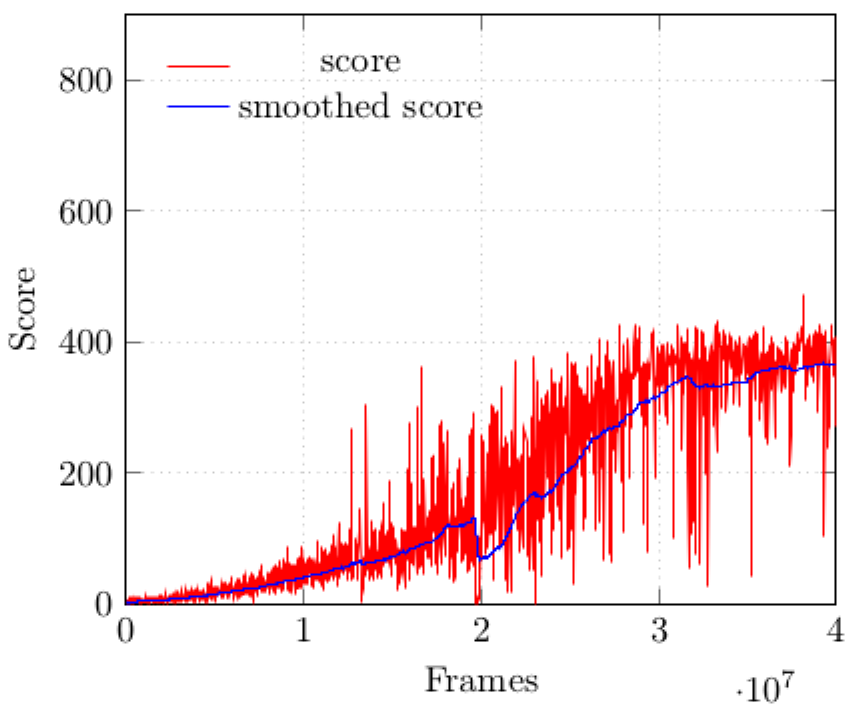


Figure 15. Instability in the learning process on the third test run of the baseline in Breakout.

My version of the A2C algorithm outperforms the well-tuned implementation of Schulman et al. (2017) [24] on Breakout significantly: their solution achieves an average score of 303 over the last 100 games of a 40 million frame training session, while my solution achieves an average score of 372.78, using the same model architecture (the baseline architecture). There are two possible explanations for this. One possible reason is that I tuned my hyperparameters for Breakout specifically while the OpenAI implementation is fine-tuned to play dozens of games as well as possible with a single set of hyperparameters. This could cause my model to play very well on Breakout but perform poorly on other games. Another possible cause for the difference is that I used general advantage estimation to estimate the advantage function for the policy gradient update (see *section 4.4.3*), and this method might be better suited for the game Breakout.

Test results for the game Qbert are shown in *figure 16*. My results on Qbert are worse than the results of Schulman et al. (2017) [24], but a similar pattern is observed as in Breakout: the baseline architecture results in slower-learning agents than A and B, and agents based on architecture B are the fastest learners. Training is less stable on Qbert than on Breakout, which is probably the result of tuning the hyperparameters on Breakout only.

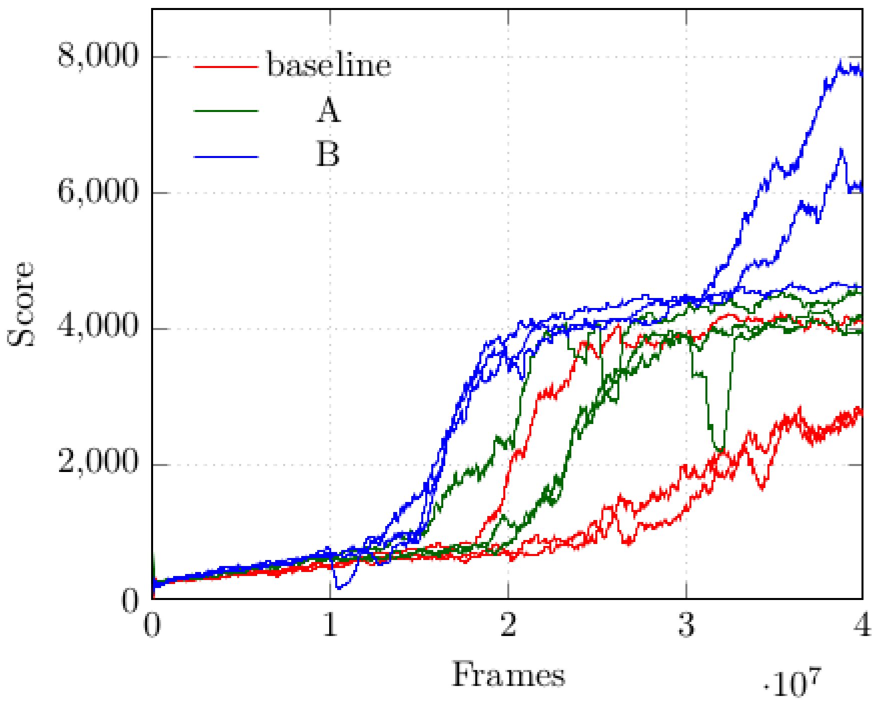


Figure 16. The learning curves of all test runs on Qbert for the the baseline, A, and B. Learning is less stable than in Breakout, since the hyperparameters were not tuned for Qbert.

The results for MsPacman are displayed in *figure 17*. In this game, B fails to learn, while architecture A still displays an increased learning ability. The only difference between A and B is the first two convolutional layers of B that A is missing, so while giving B considerable advantages in the other two games, these are the cause of the low performance in this case.

Both my baseline and architecture A-based models outperform the A2C implementation of Schulman et al. (2017) in the majority of tests. Architecture A has displayed a better learning speed than the baseline in all three games, suggesting that this A is more sample efficient in general. A has twice as many channels in its first convolutional layer as B, and the fully connected layer present in the baseline architecture before the LSTM is also omitted in A. Further testing is needed to determine to what extent these two changes contribute to the sample efficiency advantage, but it is safe to say that the fully connected layer found in the baseline before the LSTM layer is not crucial for good performance.

The extra first two convolutional layers in architecture B provide a learning advantage that dwarfs the difference between the baseline and B, but my tests show that this setup hinders learning in some environments. It is important to note that such cases can also be observed in other works. For example, in Schulman et al. (2017), the PPO algorithm generally outperforms A2C, but in the game Beamrider, A2C achieves double the performance. A possible goal for future research is to modify the extra convolutional layers of architecture B in a way that facilitates a more well-rounded performance across games.

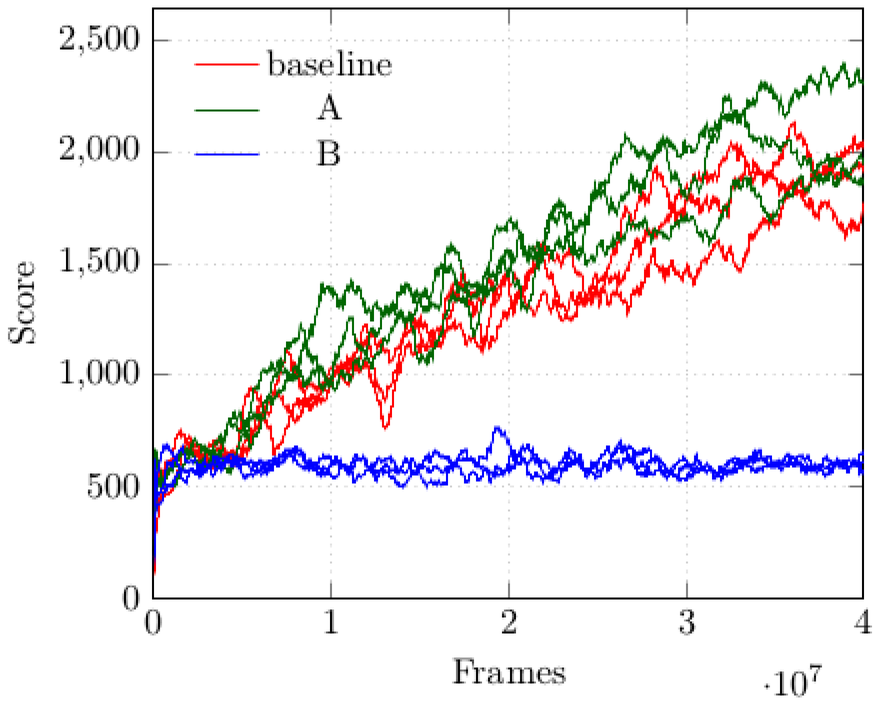


Figure 17. The learning curves of all test runs on MsPacman for the the baseline, A, and B. B fails to learn, while A outperforms the baseline.

	Breakout	Qbert	MsPacman
Schulman et al. (2017) - A2C	303.00 -	10065.70 -	1626.90 -
Schulman et al. (2017) - ACER	456.40 -	15316.60 -	2718.5 -
Schulman et al. (2017) - PPO	274.80 -	14293.30 -	2096.5 -
baseline	372.78 (7.60)	3184.33 (744.28)	1916.73 (137.65)
A	360.49 (0.64)	4238.58 (236.82)	2049.23 (224.89)
B	455.55 (11.72)	6124.67 (1548.10)	605.97 (46.49)

Table 3. The mean and standard deviation (in parentheses) of the final performance of the baseline, A, and B and Schulman et al. (2017) [24] on the tested games.

I provide the wall-clock test run times times for each game-architecture pair in *Table 4*, as measured on my testing system. The purpose of this work is to study how model architecture affects learning speed in terms of sample efficiency. Wall-clock run times are irrelevant in this context, but I still provide them for the sake of completeness. Architecture A runs slightly faster than the baseline, while architecture B is significantly slower due to its extra convolutional layers processing a high number of channels of relatively high resolution feature maps (compared to the input feature map resolutions of succeeding layers).

	Breakout	Qbert	MsPacman
baseline	4.54h	4.60h	4.53h
A	4.56h	4.52h	4.14h
B	7.47h	7.58h	7.46h

Table 4. The mean wall clock test run times in hours of the three architectures on the tested games. Architecture B requires significantly more computing time due to its first two convolutional layers. Game choice has a negligible effect on performance.

8. Conclusion

I study the effects different model architectures have on the performance of deep reinforcement learning agents in the Atari domain. My aim is to show that in addition to algorithmic advancements, improving model architectures can also lead to considerably better performing agents. After giving an overview of deep learning basics and reinforcement learning, I introduce and test two novel model architectures, comparing them to the commonly used setup of Mnih et al. (2015) [3], which I treat as a baseline. These new architectures outperform the baseline in the Atari domain in the majority of the conducted experiments. In the analysis of my test results, I describe the possible architectural reasons for the differences in performance, also outlining future directions for research.

List of Figures

List of Tables

References

[1] M G Bellemare et al. The Arcade Learning Environment: An Evaluation Platform for General Agents. *arXiv preprint arXiv:1207.4708* (21 Jun 2013).

[2] V Mnih et al. Playing Atari with Deep Reinforcement Learning. *NIPS Deep Learning Workshop* (2013).

[3] V Mnih et al. Human-level control through deep reinforcement learning. *Nature 518, 529–533* (26 Feb 2015).

[4] Z Wang et al. Dueling Network Architectures for Deep Reinforcement Learning. *arXiv preprint arXiv:1511.06581* (5 Apr 2016).

[5] M G Bellemare, W Dabney, R Munos. A Distributional Perspective on Reinforcement Learning. *arXiv preprint arXiv:1707.06887* (21 Jul 2017).

[6] V Mnih et al. Asynchronous Methods for Deep Reinforcement Learning. *arXiv preprint arXiv:1602.01783* (16 Jun 2016).

[7] M Jaderberg et al. Reinforcement Learning with Unsupervised Auxiliary Tasks. *arXiv preprint arXiv:1611.05397* (16 Nov 2016).

[8] T D Kulkarni. Deep Successor Reinforcement Learning. *arXiv preprint arXiv:1606.02396* (8 Jun 2016).

[9] P Dayan. Improving Generalization for Temporal Difference Learning: The Successor Representation. *Computational Neurobiology Laboratory, Neural Computation 5, 613-624* (1993).

[10] B Cs Csáji. Approximation with *Artificial Neural Networks. Faculty of Sciences, Eötvös Loránd University, Hungary* (2001).

[11] Y LeCun, Y Bengio, G Hinton. Deep learning. *Nature 521, 436–444* (28 May 2015).

[12] Y LeCun et al. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE* (1998).

[13] S Hochreiter, J Schmidhuber. Long Short-Term Memory. *Neural Computation 9, 1735-1780* (15 Nov 1997).

[14] A Paszke. LSTM implementation explained. Available online: https://apaszke.github.io/lstm-explained.html (30 Aug 2015).

[15] R S Sutton, A G Barto. Reinforcement Learning: An Introduction (Second edition, in progress). Available online: http://incompleteideas.net/sutton/book/bookdraft2016sep.pdf (2016).

[16] A L Strehl, M L Littman. An analysis of model-based Interval Estimation for Markov Decision Processes. *Journal of Computer and System Sciences* (2008).

[17] G Tesauro. TD-Gammon, A Self-Teaching Backgammon Program, Achieves Master-Level Play. *Neural Computation 6, 215-219* (Mar 1994).

[18] H v Hasselt, A Guez, D Silver. Deep Reinforcement Learning with Double Q-learning. *arXiv preprint arXiv:1509.06461* (8 Dec 2015).

[19] D Silver. Reinforcement Learning - Policy Gradient Methods. Available online: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html (2015).

[20] R S Sutton et al. Policy Gradient Methods for Reinforcement Learning with Function Approximation. *Advances in Neural Information Processing Systems 12, 1057–1063, MIT Press* (2000).

[21] J Schulman et al. High-Dimensional Continuous Control Using Generalized Advantage Estimation. *arXiv preprint arXiv:1506.02438* (9 Sep 2016).

[22] G Brockman et al. OpenAI Gym. arXiv *preprint arXiv:1606.01540* (5 Jun 2016).

[23] B W Mott, Stephen Anthony et al. Stella multi-platform Atari 2600 VCS emulator. Available online: https://stella-emu.github.io/

[24] J Schulman et al. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347* (28 Aug 2017).

[25] M C Machado et al. Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents. *arXiv preprint arXiv:1709.06009* (18 Sep 2017).

[26] OpenAI. OpenAI Website. Available online: https://openai.com/

[27] Hesse et al. OpenAI Baselines. Available online: https://github.com/openai/baselines (2017).

[28] D P Kingma, J Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980* (30 Jan 2017).

[29] T Tijmen, H Geoffrey. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *Coursea: Neural Networks for Machine Learning* (2012).

[30] A Saxena. Convolutional Neural Networks (CNNs): An Illustrated Explanation. Available online: http://xrds.acm.org/blog/2016/06/convolutional-neural-networks-cnns-illustrated-explanation/ (29 Jun 2016).

[31] Shi Yan. Understanding LSTM and its diagrams. Available online: https://medium.com/@shiyan/understanding-lstm-and-its-diagrams-37e2f46f1714 (13 Mar 2016).

[32] J Kvita. Visualizations of RNN units. Available online: https://kvitajakub.github.io/2016/04/14/rnn-diagrams/ (14 Apr 2016).

Appendix

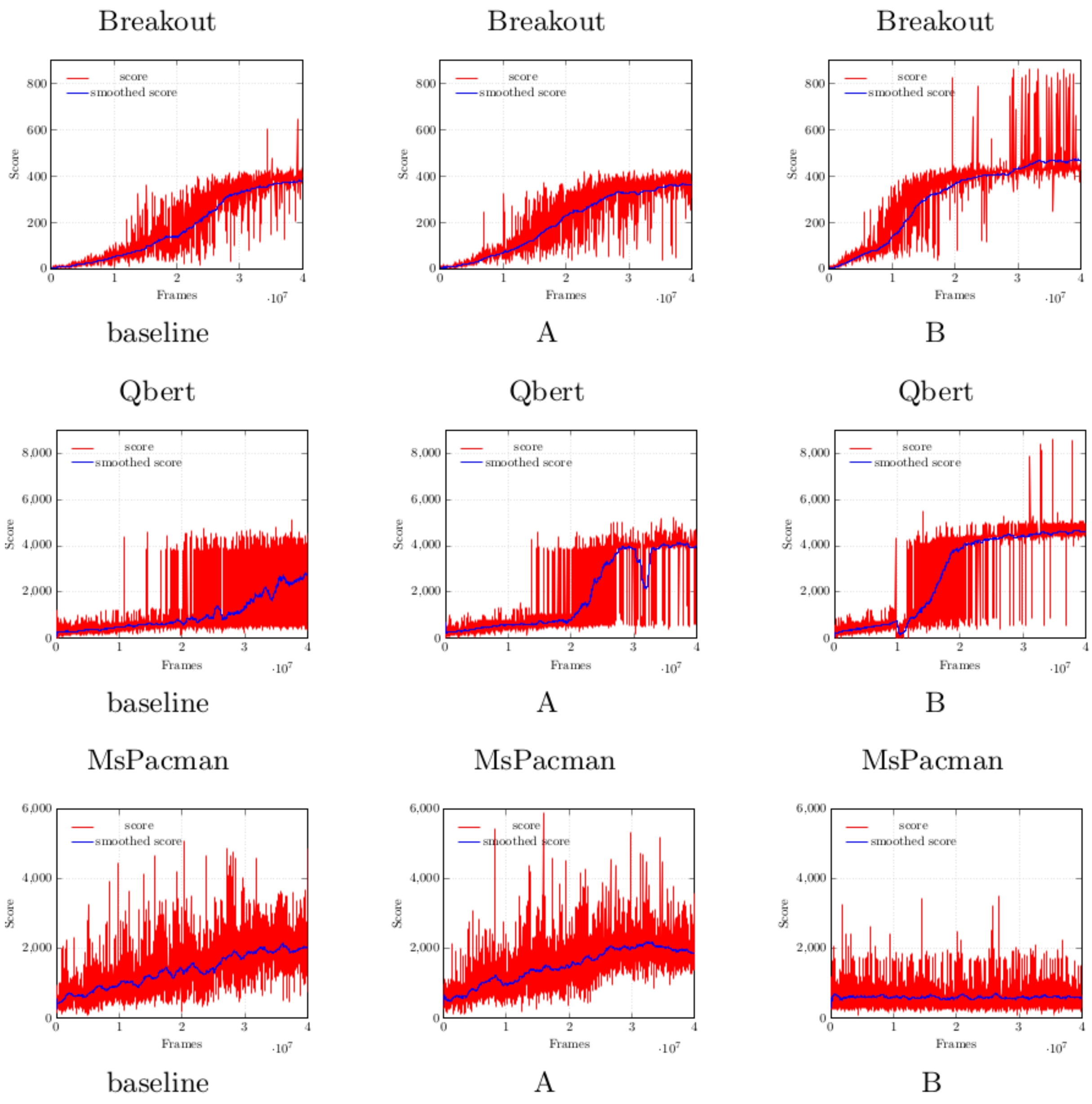


Figure 18. Raw and smoothed learning curves of the first test run for every game-architecture pair.