DEGREE PROJECT IN MATHEMATICS,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2017

Asynchronous Advantage Actor-Critic with Adam Optimization and a Layer Normalized Recurrent Network

JOAKIM BERGDAHL

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ENGINEERING SCIENCES

Asynchronous Advantage Actor-Critic with Adam Optimization and a Layer Normalized Recurrent Network

JOAKIM BERGDAHL

Degree Projects in Optimization and Systems Theory (30 ECTS credits)
Degree Programme in Applied and Computational Mathematics (120 credits)
KTH Royal Institute of Technology year 2017
Supervisor at EA SEED: Magnus Nordin
Supervisor at KTH: Anders Forsgren
Examiner at KTH: Anders Forsgren

TRITA-MAT-E 2017:81
ISRN-KTH/MAT/E--17/81--SE

Royal Institute of Technology
School of Engineering Sciences
KTH SCI
SE-100 44 Stockholm, Sweden
URL: www.kth.se/sci

Abstract

State-of-the-art deep reinforcement learning models rely on asynchronous training using multiple learner agents and their collective updates to a central neural network. In this thesis, one of the most recent asynchronous policy gradient-based reinforcement learning methods, i.e. asynchronous advantage actor-critic (A3C), will be examined as well as improved using prior research from the machine learning community. With application of the Adam optimization method and addition of a long short-term memory (LSTM) with layer normalization, it is shown that the performance of A3C is increased.

Sammanfattning

Moderna modeller inom förstärkningsbaserad djupinlärning förlitar sig på asynkron träning med hjälp av ett flertal inlärningsagenter och deras kollektiva uppdateringar av ett centralt neuralt nätverk. I denna studie undersöks en av de mest aktuella policygradientbaserade förstärkningsinlärningsmetoderna, i.e. asynchronous advantage actor-critic (A3C) med avsikt att förbättra dess prestanda med hjälp av tidigare forskning av maskininlärningssamfundet. Genom applicering av optimeringsmetoden Adam samt långt korttids minne (LSTM) med nätverkslagernormalisering visar det sig att prestandan för A3C ökar.

Acknowledgements

I would like to express my sincere gratitude for all the support from the entire SEED team including Henrik Holst and Magnus Nordin as well as my KTH supervisor Anders Forsgren. Without your help this study would not have been possible. I would also like to thank my family and friends for putting up with me during the most stressful of times. Without your endless reviewing and feedback this project could never have been completed.

Contents

Chapter 1

Introduction

In recent years, the field of deep learning has experienced great advancements in a variety of problem domains. As hardware availability and computational performance have increased, even state-of-the-art deep learning models have become accessible to the general public without the need to invest in expensive cloud computing solutions in order to train them. As a result, smaller entities ranging from research institutions to hobby enthusiasts are able to experiment with deep learning.

1.1 Electronic Arts

Digital entertainment is an industry not commonly seen as an origin of advanced research or application of modern technologies outside of rendering and computational optimization. With the desire to push the boundary of what is achievable with interactive experiences, such as video games, the digital entertainment company Electronic Arts (EA) recently took the initiative to break this misconception. An internal research and development group based in Stockholm, Sweden, was founded under the name Frostbite Labs and later renamed SEED - Search for Extraordinary Experiences Division. The motivation behind this group is to examine technologies including virtual reality, procedural generation of game worlds as well as the future of character capturing, animation and design. In addition, SEED also has a dedicated group of research and software engineers in the realm of deep learning. It is with this group the foundation for this thesis project was formed. The use cases for the technology are apparent in a multitude of areas within EA. A strong corporate incentive for deep learning is the possibility to perform expensive and time consuming tasks autonomously. An example of this is the current procedure of repeatedly testing video games during their development. Large-scale games such as entries in the Battlefield series require countless hours of play testing performed by actual humans. For the multiplayer aspect of these games, some tests are performed by using rudimentary artificial players performing random actions. This is not representative of the complex nature of actual multiplayer gameplay with many human players. Using machine learning, playtesting can be assisted with trained artificial agents with gameplay behaviour that more closely resembles that of humans. With this approach comes additional benefits such as the ability to create more human-like non-playable in-game characters such as enemies or allies that traditionally have utilized simpler state machine or behaviour tree based algorithms.

1.2 Motivation

The combination of machine learning and state-of-the-art video games is currently something relatively unexplored which serves as a motivational seed. This study focuses on the investigation of the asynchronous advantage actor-critic (A3C) algorithm developed by Google DeepMind [1]. Using only raw image data as input, A3C is a possible candidate for future application in the field of video games with the intention of creating autonomous game playing entities. This study examines the underlying structure of A3C as well as the possibility to combine the algorithm itself with research in numerical optimization and neural network layer normalization [2,3]. The core machine learning concept of this thesis is reinforcement learning which lends itself well to problems involving the constituents of video game playing.

Chapter 2

Reinforcement Learning

As its name entails, reinforcement learning is the concept of learning an underlying structure or pattern of a given problem setting using a reinforcing system - typically a system of reward and punishment. This concept is familiar to most since a majority of the learning we experience during our lifetime is greatly dependent on the notion of actions and consequences where reinforcement is dictated by the release of neurotransmitters such as dopamine or adrenaline [4, 5]. Evolutionary, this phenomenon has led to the survival of organisms. Hunting for food, migrating to more hospitable locations and breeding are instances of activities where the causal relationship between a rewarding dopamine release and the success of said activities has prolonged the existence of species [6]. The nature of the reinforcement signal is often specific to the problem at hand. A key component of reinforcement learning is the notion of exploration and exploitation [7]. Exploration is performed to slowly gather information about how the problem setting responds to certain interactions in order to understand the reinforcing reward signal and the priority of future interactions. When enough information has been parsed, it is possible to maximize the positive effects of the reinforcing signal by exploitation. In reality, with enough engineering many situations encountered by a human could be interpreted as a reinforcement learning setting [7]. Learning how to cook, ride a bicycle or play tennis are all valid reinforcement learning situations.

2.1 The Reinforcement Learning Setting

In general, a reinforcement learning problem can be partitioned into a few key components. Most of these components are essential in order to formally define a reinforcement learning setting but their shape or form varies from problem to problem. The following sections explain the properties of a reinforcement learning problem and their individual variations. In the reinforcement learning problems based on the real-life situations explained in the previous section, we as humans are the actors performing exploration and experience gathering in order to understand them. The entity synonymous to the method or algorithm used to solve a reinforcement learning problem will from here on be referred to as the *agent*. The agent interacts with an *environment* using actions described by and limited to the so called *action set*. When performing an action, the agent observes the change in the environment by sampling its *state space*. As well as the observation, the agent also receives a scalar response in the form of a reinforcing *reward signal*. In a computational setting, the agent follows a discrete timeline and interacts with the environment until a terminal state is reached - a state in which the agent is deemed to have failed its task. A simplified agent-environment interaction scheme is presented in Figure 2.1.

A Reinforcement Learning Setting

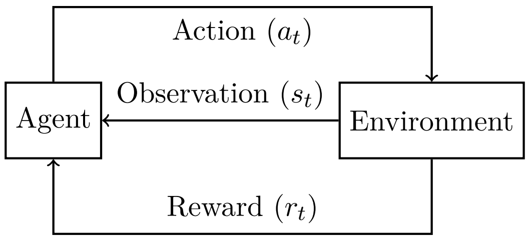


Figure 2.1: Simple reinforcement learning flowchart of the interaction between the agent and the environment at time t. When performing action a_t, the agent receives an observation s_t and a scalar reward r_t.

Representation of Time

It stands to reason that reinforcement learning problems in a simulated and programmatic setting are solved with a numerical approach using a discrete timeline. For some point $t \in \{0, 1, 2, \ldots, T_{max}\}$ in time, the agent is procedurally learning until it is fully trained and performing according to the capabilities of the chosen reinforcement learning algorithm. The timestep T_{max} represents the time horizon for when the algorithm terminates. For certain problems the agent may need multiple attempts ending in failure to get an understanding of how to solve them, much like falling over with a bicycle when learning how to ride it. Problems like these are seen as *episodic* where an episode corresponds to the time period from the initialization of the environment to the timestep where the environment signals to the agent that no further actions can be performed. The timeline for episodic problems are partitioned into sequences where the agent perform consecutive attempts at exploring the environment [7]. One example of such a problem is the one-dimensional cart-pole or inverted pendulum balancing problem [8]. Here, the task is to balance an inverted pendulum on a cart on which a force can be exerted in either the left or right direction along the x-axis. When the pendulum swings outside of a specified angular boundary or the cart leaves a designated spatial interval the environment resets and the agent has to balance the pendulum again. A collection of multiple episodes is often referred to as an *epoch*. Using either pure timesteps, episodes or epochs is viable when training a reinforcement learning agent depending on the type of problem investigated.

Environment

How reinforcement learning problems are separated primarily boils down to the enclosed universe in which the properties of the specific problems themselves are defined - the environment. Instances of problem environments can be the surroundings of an autonomous vehicle, the collection of joints of a walking robot or a game of go [1,9]. An important feature of the environment is its ability to convey information about its current state which is the internal representation of what is actually happening when it is interacted with. As this information may be abstract, it is up to the designer of a possible solution method based on the environment to define it and use it in a purposeful way. States for a game of go could be the exact position of every piece on the board. The actual internal description of the current state of an environment resides in the state space $\mathcal{S}$.

State Space

The information the agent receives when observing the environment is represented by a state $s_t \in \mathcal{S}$. The state space contains every state possible in the environment. However, not all of these states may be observable by the agent. The nature of the state space dictates the design of the actual reinforcement learning algorithm as it may be infeasible to experience every observable state of the environment. For a game of go with a boardsize of 19×19, the number of possible legal game piece positions amounts to more than 10^{170} combinations [10]. Observations made in the environment may only represent a small part of what is actually occurring within the environment. The image data captured by a camera is an example of an observation where what is located out-of-frame may be unknown. If the task of the agent is to play a video game against a human player with a game videofeed as input it would be unfair to the human player if the agent could partake in information unavailable to the human. For these reasons, it is important to make a distinction between states and observations. The states that the agent actually can experience exist in the observation space which is merely a subset of the state space. When training, the agent implicitly explores the state space via the observation space. Given enough state space information the agent slowly learns to infer the consequences of its actions based on their impact on the environment. Every action a_t at time t available to the agent interacting with the environment whilst observing s_t is contained in the action set $\mathcal{A}(s_t)$.

Action Set

The most basic type of action set contains a finite, countable number of actions. This is sometimes referred to as a *single-action* setting as only one member of the action set can be sampled by the agent at time t. For the aforementioned cart-pole balancing problem, the action set is defined by $\mathcal{A} = \{left, \mathit{right}\}$. Expanding this set for some environment allowing for two degrees of freedom, enabling movement along the y-axis, is formulated by

$$\mathcal{A} = \{left, \mathit{right}, \mathit{up}, \mathit{down}\}. \tag{2.1}$$

Typically, each action is unique and enumerated following a static order such that actions are easily identified. If the resulting action from the combination of two or more actions too is unique, it is usually represented as a separate entry in the action set. Combining the actions *up* and $left$ in (2.1) would result in a diagonal motion which could constitute a fifth element in the action set. In some situations during training, there may be an incentive to not perform any action at all. This lack of interaction is represented by the so called *no-op* action. This is the conscious decision to avoid any form of interaction which could yield unwanted results whilst the state of the environment is changing. When designing a solution model to a reinforcement learning problem, it is important to understand the properties of the action set and only let it include actions meaningful to the environment. In real life scenarios, binary actions are not always adequate for situations where fine grained control is needed. An autonomous vehicle needs to be able to adjust its velocity using more than full left or right steering and full or no throttle. A problem like this relies on a more intricate *continuous action space*, which dictates the degree of application of a certain action. In this thesis, only discrete action sets are investigated in a single-action setting.

Policy

When observing state s_t, the agent has to be able to infer which action residing in $\mathcal{A}$ it should perform. The strategy of choosing this action is represented by the policy $\pi(s_t)$ which can be viewed as the mapping from state space to the action set according to $\pi : \mathcal{S} \to \mathcal{A}$. Without enforcing exploration of the environment, the agent might never try to perform actions that seem unintuitive but may yield high rewards. Therefore, the action selection process benefits from being stochastic. In practice, the agent uses the policy to sample actions from $\mathcal{A}$ based on their individual potentials in generating future reward. Hence, the policy defines the probability distribution over all elements in $\mathcal{A}$ according to

$$\pi(s_t) = \left[P(a_1|s_t), P(a_2|s_t), \ldots, P(a_{M-1}|s_t), P(a_M|s_t)\right], \tag{2.2}$$

where M is the number of actions in the action set. Leveraging the distributional properties of the policy, the agent samples actions from the action set leading to actions accounting for low probabilities being stochastically selected which helps with exploration [1]. Shaping these policies is the basis of reinforcement learning and the large variety of solution methods available dictates the nature of this process [7]. During training, the relationship between input states and policies are iteratively forged using the agent's reward signal feedback.

Reward Signal

Whilst interacting with the environment, the agent needs some quantifiable, reinforcing response to progress with its training. In a simulated setting, this response is composed of a scalar reward, which is transmitted to the agent after every interaction with the environment. For some point t in time, the reward $r_t \in \mathbb{R}$ represents the agents momentaneous performance in the environment. In practice, this scalar reward signal is composed of positive and negative values corresponding to both reward and punishment. Returning to the scenario of the autonomous vehicle, the agent controlling it may be positively rewarded for each meter successfully driven without crashing or punished for each time it drives outside the boundary of its designated road. As the task of the autonomous vehicle is to move, it could also be punished by standing still. The measure of reward is essential when training a reinforcement learning agent. Careful engineering of the numerical reward values for each meaningful event in an environment is referred to as *reward shaping.* As the reward is a function of the current state of the environment and the action selected by the agent, it can be represented by $R_t : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Accumulating consecutive rewards according to

$$R_{total} = \sum_{i=1}^{\infty} r_i \tag{2.3}$$

provides little information to the agent about future, potential rewards that could be useful when searching for better policies. If there is no termination step T_{max}, this metric can only really be used in trivial, episodic problem as the sum is infinite. In non-trivial problems, the discounted future reward obtainable by the agent in state s_t is a more interesting measure formulated by

$$R_t(s_t, a_t) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{2.4}$$

where $\gamma \in (0, 1)$ is a discounting factor. This sum represents the present value of possibly receiving rewards at future points in time discounted by γ [7]. Rewards too far into the future are evaluated to zero given $\gamma < 1$. This effect provides a limit to the number of timesteps for the agent's forward view and it is evident that (2.3) corresponds to the special case when $\gamma = 1$. Environments may have a dense reward signal with rich, high-frequent reward feedback or a sparse signal where the agent is unrewarded for many timesteps resulting in a problem more difficult to solve.

2.2 State-Value and Action-Value Functions

The agent interacts with the environment in order to incrementally gain experience. By these interactions, the agent learns properties inherent to the environment in its quest to find policies that maximizes its received reward described by (2.4) in a long-term perspective. In the training phase, the agent slowly constructs causal relationships between perceived states and the most beneficial actions it should perform. For the agent to be successful, it can not merely rely on random exploration when acting in the environment. If this is the case, the agent will not observe a large enough set of environment states to understand the entire problem setting. To circumvent the obvious risk of finding a local extreme point with regards to the agents reward measure, it is important that the agent has a viable exploration strategy in order to experience states that are initially rare during training. Consider a setting where the agent follows a discrete timeline $t \in \{0, 1, 2, \ldots, T_{max}\}$. In order to learn, the agent uses the reward signal (2.4) as a measure of success of a performed interaction and its manifested result in the environment. There is always a limit to the expected reward the agent can receive when transitioning from one state to the next given an optimal action. This is quantified by the action independent state-value function

$$V(s) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right], \tag{2.5}$$

which maps elements of the state space to scalars according to $V : \mathcal{S} \to \mathbb{R}$. If other actions than the most optimal are performed, the state-value function (2.5) needs to be expanded. The value of a state given a specific action is therefore defined by the action-value function

$$Q(s, a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a\right], \tag{2.6}$$

which similar to the state-value function maps state-action pairs to real valued scalars $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$.

2.3 Solution Methods

For discrete and finite state and action sets, the general reinforcement learning problem can be viewed as a finite Markov decision process (MDP) due to the fact that the probability of transitioning into some state s_{t+1} only depends on the previous state s_t and action a_t. Even if this assumption may be formally weak for some reinforcement learning problems where the Markov property may not always hold, it is useful for any problem where estimation of future state transitions is important. In non-trivial reinforcement learning environments the probability for transitioning between two consecutive states s_t and s_{t+1} with some action a_t is generally unknown. As a result, the dynamics of the problem environment have to learned. This type of problem requires a solution method referred to as model-free as opposed to model-based where the state transition probabilities are explicitly predefined. It stands to reason that a reinforcement learning problem is solved by finding the policies $\pi(a_t|s_t)$ maximizing (2.5) where the optimal state value is defined by

$$V^*(s) = \max_{\pi} V^{\pi}(s). \tag{2.7}$$

By extension, these optimal policies also yield an optima for (2.6). For actions residing in $\mathcal{A}$, the Bellman optimality equations for the state and action value functions can be formulated as

$$\begin{aligned} V^*(s) &= \max_{a} \mathbb{E}[R_{t+1} + \gamma V^*(s_{t+1})|s_t = s, a_t = a], \\ Q^*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})|s_t = s, a_t = a]. \end{aligned} \tag{2.8}$$

Recursively solving (2.8) by searching for actions with corresponding state transitions to find optimal policies in the MDP may be infeasible for problems with large state spaces. By instead directly substituting $V^*(s)$ and $Q^*(s,a)$ with viable function approximators and use them to estimate the state- and action-values it is possible to create an algorithm that iteratively infers the policies corresponding to the current optimal state-action values. A common choice for such an approximator is the *neural network*. Using a neural network with parameters or weights θ, the optimal state and value functions (2.8) can be parameterized by

$$\begin{aligned} V^*(s) &\approx V(s;\theta), \\ Q^*(s,a) &\approx Q(s,a;\theta). \end{aligned} \tag{2.9}$$

Instead of inferring optimal policies from the state- and action-value functions one may also parameterize the policy function itself according to

$$\pi(a_t|s_t) \approx \pi(a_t|s_t;\theta), \tag{2.10}$$

which is central to this thesis as further explained in Section 4. Here, $\pi(a_t|s_t)$ represents the conditional probability for a specific action $a_t \in \mathcal{A}$ given state s_t. The foremost argument for using neural networks is their ability to perform generic feature extraction from input data. As in general machine learning, the performance of a neural network is gauged with an *error function* measuring the difference between the input data and the expected output data. When the network performs well, the error function yields small values and vice versa. The features extracted by the network directly correspond to the properties in the input data that has the largest effect on the error function. This eliminates the need for *feature engineering* which is the time-consuming task of manually picking important features from the input data. Further, neural networks have the property of being differentiable making gradient based methods for changing the network parameters θ possible, enabling incremental adjustments to the input-to-output mapping of the network.

2.4 Feedforward Artificial Neural Networks

Being a function approximator, the fundamental action of an artificial neural network (ANN) is to map input to some output set. As a continuation of the reinforcement learning setting, this input may reside in $\mathcal{S}$ with a resulting output being approximations of the state- and action-value functions described in Section 2.2. Internally, an ANN is composed of connected layers of nodes referred to as neurons. A common class of ANN layers is the fully connected one where each layer neuron is connected to every neuron of the previous layer [11]. These connections are weighted in such a way that data flow in the network can be controlled down to a neural level. In a feedforward neural network, data is only propagated in the direction from the first input layer to the last output layer. One iteration of computing the output from the network is commonly referred to as a *forward pass*. The output of the entire ANN can be seen as the result of a chain of functions f_i where each function operates on the output of its predecessor f_{i-1} according to

$$f(\mathbf{x}) = (f_n \circ f_{n-1} \circ \cdots \circ f_2 \circ f_1)(\mathbf{x}), \tag{2.11}$$

for a network of n layers. The weights of the connections to layer i can be expressed as a real-valued matrix $\mathbf{W}_i$ which upon multiplication with some input allows for or cancels flow in a given neuron-to-neuron connection. Given some intermediate layer input $\mathbf{z}$ the function f_i for layer i is formulated as the linear transformation

$$f_i(\mathbf{z}, \mathbf{W}_i, \mathbf{b}_i) = \mathbf{z}^T \mathbf{W}_i + \mathbf{b}_i, \tag{2.12}$$

where $\mathbf{b}_i$ is the bias of layer i. The bias allows for translation of the product $\mathbf{z}^T \mathbf{W}_i$ and corresponds to the inherent difference between the model itself compared to the best hypothetically possible model [11].

A problem with using only linear transformations is the inability of the network to facilitate non-linear output. In order to increase the generalization ability of (2.12), the output of the linear function is instead passed through fixed, non-linear and elementwise *activation functions* g [11]. Among others, this can be the rectified linear-unit function (ReLU), sigmoid function σ and the hyperbolic tangent function tanh. With this, the intermediate network function (2.12) is redefined by

$$f_i(\mathbf{z}, \mathbf{W}_i, \mathbf{b}_i) = g(\mathbf{z}^T \mathbf{W}_i + \mathbf{b}_i). \tag{2.13}$$

For a multilayered network, the parameters of a given layer i can be formulated as $\theta_i = [\mathbf{W}_i, \mathbf{b}_i]$. Hence, when referring to the set of parameters describing the entire network it can be expressed as $\boldsymbol{\theta} = [\theta_1, \theta_2, \ldots, \theta_n]$. A neural network model is trained by updating $\mathbf{W}_i$ and $\mathbf{b}_i$ for each layer. The standard procedure of performing such an update is via error back-propagation methods such as stochastic gradient descent (SGD), which is further examined in Section 4.4. Only the output of the final network layer has a predetermined structure and the layers between them are generally incomprehensible to an external observer. Therefore, these layers are commonly referred to as *hidden*. With a change of notation, the hidden layer $\mathbf{h}_{i+1}$ is defined by the non-linear output and parameters of the previous layer according to

$$\mathbf{h}_{i+1} = g(\mathbf{W}_i^T \mathbf{h}_i + \mathbf{b}_i). \tag{2.14}$$

The notion of *deep* learning stems from the use of deep neural networks with many hidden layers. The limit for when a neural network based method becomes one of deep learning is undefined and more attributed to the renaissance of the use of neural networks during the beginning of the 21st century. The combination of reinforcement learning and deep learning is referred to as *deep reinforcement learning.*

Chapter 3

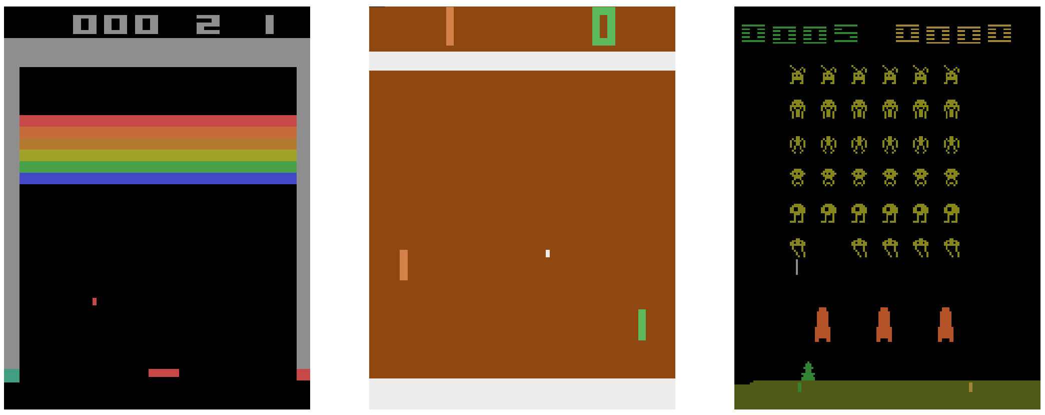
Playing Atari 2600 Games Autonomously

As the aforementioned reinforcement learning setting implies, the number of constructable reinforcement learning problems are infinite. Any problem with a viable reward signal may constitute a valid reinforcement learning problem. Imagine designing a general agent with the ability to solve any given reinforcement learning problem, discrete as continuous with any definable action set and with any state space. This task is incomprehensibly large and impossible to solve in feasible time with current technologies. One could argue that if this problem was to be solved, the first instance of artificial general intelligence (AGI) could be created. However, reinforcement learning is merely a small toolset on the path towards achieving AGI. When designing a reinforcement learning agent, it is desirable to use a set of standardized problems broad enough to see whether said agent is able to generalize its solution strategies to solve problems that are similar but not necessarily equal. Currently, a common set of problems used by reinforcement learning researchers is the one of Atari 2600 games [12–14]. Inspired by this, the core reinforcement learning problem of this study too is autonomously playing Atari 2600 games. In this section, the Atari environment is presented with its properties applicable to reinforcement learning and how it can be decomposed to make it compatible with a reinforcement learning algorithm.

3.1 Atari 2600 Video Computer System

The Atari 2600 Video Computer System (VCS) was one of the best selling consoles of its time after its release in 1977. A natural intention in developing video games for the system was to create unique experiences in order to stay ahead of the competition. As a result, the Atari 2600 games catalog contains a diverse group of games ranging from sports titles to shooting games. By extension, these games ought to serve as a good playground of problems with a broad set of characteristics, making it usable in the domain of reinforcement learning. Ideally, a reinforcement learning method should be general enough to be able to play a multitude of these games without any algorithmic modification. In the Atari 2600 games catalog are games where there is an immediate score response when performing an action such as in Space Invaders (1978), where the player fires projectiles at an incoming alien invasion. Other games require a measure of planning as in the case of Montezuma's Revenge (1983), where the player needs to pick up items in one stage of the game and use it in another to progress in the adventure setting of the game. Specifically, Montezuma's Revenge has proven to be a difficult game to solve for current state-of-the-art reinforcement learning methods unless the reward signal is shaped in such a way that complex long-term rewards are accounted for, and advanced exploration strategies are used [15]. In this study, three games are investigated. These are Breakout, Pong and Space Invaders. Example screenshots from these games are displayed in Figure 3.1 with their respective game descriptions.

Atari 2600 Games



(a) Breakout (b) Pong (c) Space Invaders

Figure 3.1: Three instances of Atari 2600 games used in this study. 3.1a Using a paddle, the goal in Breakout is to catch and strike a ball in order to break bricks in the upper half of the screen. Missing the ball results in the loss of one out of five extra-lives. 3.1b In Pong, the player controls the green, rightmost paddle in a game of rudimentary table tennis. When the opponent (left paddle) misses the ball and it passes the left boundary of the playing field, the player scores and vice versa. 3.1c In Space Invaders, a vehicle on the lower half of the screen is controlled by the player. The goal is to shoot down incoming aliens that shoots back at the player. The score count increases when destroying alien ships as well as a special pink saucer that randomly travels in the top boarder of the play area. If the player is hit by the firing aliens, the player loses one of five extra-lives.

3.2 Action Set

A benefit of the Atari 2600 VCS is its simple game controller composed of an 8-directional joystick and a fire button allowing for eighteen possible action and action combinations. These actions are displayed in Table 3.1.

Atari 2600 Controller Action Set

No-Op	Fire	Up
Right	Left	Down
Up + Right	Up + Left	Down + Right
Down + Left	Up + Fire	Right + Fire
Left + Fire	Down + Fire	Up + Right + Fire
Up + Left + Fire	Down + Right + Fire	Down + Left + Fire

Table 3.1: The actions allowed by the Atari 2600 VCS controller [16]. Note the combinations of actions in lower rows of the table.

Constructing $\mathcal{A}$ for a given Atari game is done by merely extracting the subset of actions in Table 3.1 relevant to that game. The individual action sets for Breakout, Pong and Space Invaders illustrated in Figure 3.1 are shown in Table 3.2.

Individual Action Sets

Breakout	No-Op	Fire	Left	Right		
Pong	No-Op	Up	Down			
Space Invaders	No-Op	Fire	Left	Right	Left + Fire	Right + Fire

Table 3.2: Action sets for Breakout, Pong and Space Invaders. The *Fire* action in Breakout corresponds to resetting the game after an in-game life has been lost after missing the ball. If this action is not performed, the ball will not reappear.

3.3 Image Data and State Space

For the Atari environment, each game screen constitutes a state s in $\mathcal{S}$. Being a system from the 1970's, there are some hardware limitations to consider. The resolution supported by the Atari 2600 is generally undefined and the maximum, possible resolution of 192×160 pixels deliverable by the system is only acheivable by exploiting the method with which areas of the screen is rendered [17]. In a machine learning setting, the Atari environment is generally used with the Arcade Learning Environment (ALE) based on the Stella emulator. In ALE, the resolution is scaled to 210×160 pixels independent of the game [16]. Each frame is rendered with three color channels corrsponding to the RGB color space, effectively making a single screenshot from the emulator $210 \times 160 \times 3$ pixels. The size of these images makes the image data processing unwieldy and expensive in a real-time setting. This is further reinforced by the fact that the Atari 2600 performs 50 to 60 frame updates per second. Using ALE in conjuction with a reinforcement learning model benefits from going through a preprocessing step before image data is fed to the model. First, the color channels along the depth dimension of the images are reduced by performing a luminosity mapping, generating grayscale images. Secondly, as the graphical fidelity of these games is low as only four colors can be displayed concurrently from their 128-color palette, not much information is lost if the game images are scaled down. Typically, the images are downscaled to 84×84 [1, 12, 13]. Following these preprocessing steps, the produced grayscale game screenshots or states have the dimensionality of 84×84. For Breakout, Pong and Space Invaders, the entire gameplay areas are static and always visible. As a result, these games are fully observable.

3.4 Previous Solution Methods

The Atari problem domain has been used in a multitude of studies in the deep reinforcement learning field. Mnih et al. published an off-policy solution method based on an amalgamation of deep learning and Q-learning named deep Q-network (DQN) in 2013 [14]. Improving upon the ground work by Bellemare et al. which displayed the usability of the Atari domain with a set of reinforcement learning algorithms such as SARSA(λ), the DQN model proved to successfully play a set of Atari games even surpassing the performance of a human player in some of them [13, 14, 16]. In 2015, Schulman et al. presented an iterative method for optimizing policies with monotonic improvements by defining a trust region for the policy updates based on constraining their Kullback-Leibler divergence [12]. The method named trust region policy optimization (TRPO) proved to be robust whilst achieveing scores in Atari games comparable to the DQN method but also solve simulated robotic locomotion problems. Common for these approaches is the use of image data from which necessary features are extracted using convolutional neural networks (CNN). Generally, the complexity of the computations required for handling image data results in slow training. This is commonly alleviated by performing said computations on graphics processing units (GPU) which are designed to efficiently perform parallel operation exectutions using multiple cores. Instead of depending on high-end GPUs, Mnih et al. proposed the A3C reinforcement learning method inspired by the Gorila RL framework previously developed by Google DeepMind [18]. The core motivation of A3C was it parallelizability by distributing the training process over separate central processing unit (CPU) cores resulting in a data efficient model outclassing the training performance of previous methods and severely decreasing the required training time [1, 18].

Chapter 4

Asynchronous Advantage Actor-Critic

4.1 Asynchronous Training with Multiple Learner Agents

The high computational efficiency of the A3C algorithm originates from its use of separate learner agents exectuted in parallel [1]. These agents asynchronously update the parameters of a central neural network model using a stochastic gradient descent based optimization method. Each learner agent interacts with a separate instance of the environment by inferring actions from an independent, local copy of the central network model. When an agent finishes an episode in its environment instance, its local network copy is overwritten with the current state of the central network model. At each episode-termination, the central network model is updated using the parameters of each learner agents network instance. In essence, the A3C method can be viewed as an ensemble of many simple reinforcement learning agents similar to DQN. It is their collective influence on the central network that accounts for its performance. As the learner agents have the ability to explore their environments independently, the training data becomes diverse. Given the multi-threaded nature of A3C, it is easy to distribute its main training process over separate cores of a CPU. Given adequate hardware, the parallelized training can experience a speed-up by simply increasing the number of training threads in the algorithm [1]. As CPU based compute in general is more readily available as opposed to expensive GPU clusters, the A3C algorithm can be deployed with most cloud computing services with relative ease. These properties aside, it is however suggested in the original article that the training potentially can be accelerated with the help of GPUs which is taken into consideration in this study. See Section 6.3 for further notes.

4.2 Actor-Critic and Advantage

Reviewing the concepts of Section 2.1, general reinforcement learning solution methods are based on the policy and the state and action value functions in various combinations. Methods based on learning using the value functions are referred to as critic-only. These methods initially try to approximate optimal value functions and then use them to infer optimal policies. A few examples of critic-only methods are dynamic programming, temporal difference (TD) learning and eligibility traces [19]. Instead of using only the value functions, it is possible to solve reinforcement learning problems by searching for optimal policies directly in the policy space. This type of method is referred to as actor-only. Combining the policy and state-value functions, one can also formulate an *actor-critic* setting as the basis of a reinforcement learning method which is central to the A3C model. The notion comes from the interaction between an actor controlling action selection and a critic quantifying the action selection performance of the actor. In A3C, the actor is represented by the policy $\pi(a_t|s_t;\theta)$ and the critic is an estimate of the *advantage function* $A(s,a;\theta)$. The advantage function is typically defined by

$$A(s_t,a_t) = Q(s_t,a_t) - V(s_t), \tag{4.1}$$

which evaluates the *advantage* in performing an action that may not be the most optimal corresponding to $V(s_t)$. For A3C, the advantage function is instead formulated as the difference between the expected future reward when performing action a_t in state s_t and the actual reward that the agent receives from the environment [1,20]. This is represented by

$$A(s_t,a_t;\theta) = \sum_{i=0}^{k-1} \gamma^i r_{t+1} + \gamma^k V(s_{t+k};\theta) - V(s_t;\theta), \tag{4.2}$$

where k is upper bounded by T_{max} and $V(s_t;\theta)$ is the ANN-approximated state value function parameterized by θ. Using the reward signal defined by (2.4), the advantage function formulation can be simplified to

$$A(s_t,a_t;\theta) = R_t(s_t,a_t) - V(s_t;\theta). \tag{4.3}$$

If training is successful, the value of the advantage function should ideally converge according to

$$\lim_{t\to\infty} (R_t - V(s_t;\theta_v)) = 0, \tag{4.4}$$

which represents a point in time where the model estimates a state-value that is equal to the actual received reward. In practice, the parameterization of π and V may be done with two neural networks with the separate parameter sets θ and θ_v respectively. It is also possible to use a single network with shared initial layers but with two separate output layers corresponding to π and V described later in Chapter 4. From now on, the policy and state function are defined by these separate parameterizations according to $\pi(a_t|s_t;\theta)$ and $V(s_t;\theta_v)$.

4.3 Convolutional Neural Network

When using imagery as input data, a preferable neural network model for extracting features within said data is the convolutional neural network. Its name originates from its use of the convolution operation between a set of convolutional kernels and the image data, as opposed to the matrix multiplications seen in Section 2.4. In this context, an image is represented by a tensor of rank 3 containing the scalar intensities for each pixel with dimensions width, height and depth where the latter corresponds to the number of color channels in the image. The convolutional kernels are represented by rank 2 tensors with width and height dimensions, generally much smaller than those of the input data. Convolving the input image with a given kernel amounts to calculating the inner product between the kernel tensor and a slice of equal size in the input data. This procedure is repeated by moving the slice over the image with equidistant steps referred to as *strides*. The values from the products generates a two-dimensional *feature map* where each element is a scalar representation of a small region in the input data with respect to the spatial relations between the image pixels in the original image slice. By using multiple different kernels, the generalization ability of a convolutional layer is increased with the number of resulting activation maps. When speaking of convolutional layers, one usually refers to its output feature maps as its *output channels*. Analogous, the stack of kernels operating on the input data is referred to the *input channels* of the layer. The convolutional layer output is generally passed through the ReLU function defined by

$$g(\mathbf{z}) = \max\{0, \mathbf{z}\}. \tag{4.5}$$

The activation function works as a threshold for when a given feature in the feature map is registered. In a typical Atari 2600 game, there are plenty of screen-space objects moving between each updated frame. In order to train an agent how to play a game of Pong it needs to understand the motions of the ball as well as the paddle it is controlling. By feeding only a single emulator image to the agent it would lack the temporal information needed to infer the velocities of these in-game objects. The solution to this problem is to instead feed the agent with a stack of consecutive images. In the original A3C formulation each input observation s_t is a stack of four greyscale images such that $s_t \in \mathbb{R}^{84\times84\times4}$. The network architecture used in A3C utilizes two consecutive convolutional layers as seen in Figure 4.1 which corresponds to the network in the original DQN model [14].

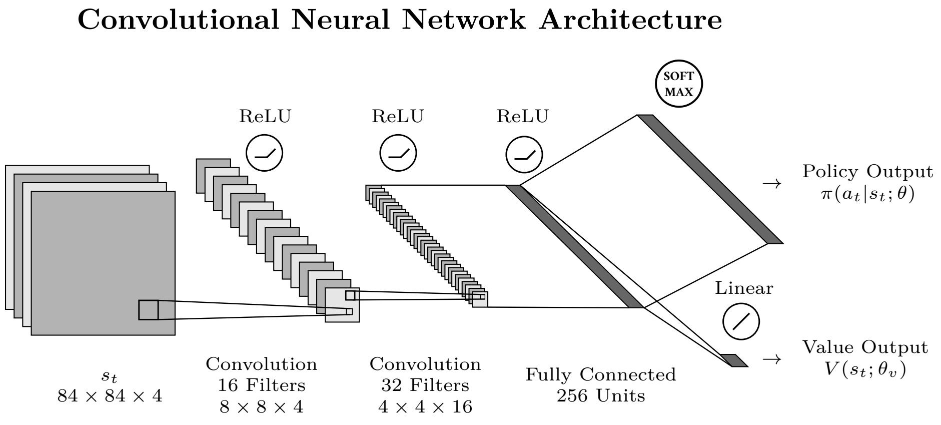


Figure 4.1: Illustration of the convolutional neural network used by Mnih et al. in the A3C model [1]. The first convolutional layer convolves input states s_t in the form of a stack of four consecutive, greyscale 84×84 frames from the Atari environment with 16 filter kernels of size 8×8 and a stride of 4, resulting in 16 feature maps of size 20×20. The second convolutional layer convolves the 16 output channels of the previous layer with 32 kernels of size 4×4 and a stride of 2, producing feature maps of size 9×9. The third fully connected layer of 256 units is the final layer shared among the θ and θ_v parameter sets. Each of these three initial layers are passed through the non-linear ReLU activation function.

The 16 kernels used for convolving the input in the initial hidden layer is of size 8×8 with a depth of 4 and a stride of 4. The second hidden layer performs convolutions with 32 kernels of size 4×4 with a depth of 16 as well as a stride of 2. The third hidden layer is fully connected with 256 neurons. The policy output of the model corresponds to the normalized, elementwise values of the *softmax* function operating on the output of the fully connected layer, yielding a probability distribution over the actions in $\mathcal{A}$. The value function estimation output is represented by a single, linear output deduced from the fully connected layer. Each of the three hidden layers of the network corresponds to the output of the ReLU activation function. An entire instance of the CNN model contains $6 \cdot 10^5$ parameters, which are update during the course of training.

4.4 Optimization Problem

In the case of the Atari 2600 game domain, the reinforcement signal is constituted by the score or reward accumulated during state transitions, i.e. if an agent performs an action a_t within the game such that the current game score increases, it receives some measurable reward in the transition from state s_t to s_{t+1}. As mentioned in Section 4.2, the A3C algorithm is based on the actor-critic paradigm with $\pi(a_t|s_t;\theta)$ as the actor and the advantage function estimation $A(s_t, a_t; \theta_v)$ as the critic. Policy and state-value outputs are generated by the final layers of the CNN presented in Figure 4.1. The purpose of the training phase is to find optimal parameter sets θ and θ_v yielding policies that maximize the future reward of the agent. Given the complex and non-linear nature of the CNN approximation of π, finding optimal policies leads to a non-convex optimization problem. Using the formulation by Mnih et al., the scalar objective function for the parameterized policy function can be derived as

$$f(\theta, \theta_v) = \log \pi(a_t|s_t;\theta)(R_t - V(s_t;\theta_v)), \tag{4.6}$$

where the logarithm is computed elementwise over the probabilities of the policy vector. Given the properties of (4.6), it is difficult to fully understand its range of outputs due to the nature of π being a discrete probability distribution produced by the final softmax function as seen in Figure 4.1 as well as the behaviour of the advantage estimation $R_t - V(s_t;\theta_v)$ which may be positive or negative. There is a risk of $\log \pi(a_t|s_t;\theta)$ decreasing unboundedly to $-\infty$ if at least one probability in the policy converges to zero which is disruptive in a numerical optimization setting due to the risk of underflow. As it is impossible to sample actions corresponding to zero-valued probabilities, a policy of this kind results in a less non-deterministic action selection behaviour by the agent. In an extreme case, only a single member of the policy may constitute the entirety of the probability mass of the distribution resulting in all other policy members being zero-valued. An trivial example of such a policy can be formulated as

$$\pi(a_t|s_t;\theta) = [0, 0, 0, 1], \tag{4.7}$$

for an action set with 4 actions. This is referred to as a fully *deterministic policy* as the agent is only able to consistently sample a single, predetermined action. As mentioned in Section 2.2, exploration is important to successfully solve a reinforcement problem. With the occurrence of deterministic policies, exploration becomes more difficult and the risk of overfitting the model to a small subspace of $\mathcal{S}$ increases.

4.4.1 Introducing Entropy

To incentivize the avoidance of less non-deterministic policies, the objective function needs to enforce a certain measure of stochasticity in the action selection strategy of the agent. This can be achieved with the help of the *entropy* of the policy distribution π [1]. Using the knowledge of the advantage function and the range of

$\log \pi(a_t|s_t;\theta)$, the objective is to maximize (4.6) with respect to the model parameters θ and θ_v. For a given policy $\pi(a_t|s_t;\theta)$, the Shannon entropy $H(\pi(a_t|s_t;\theta))$ is formulated by

$$H(\pi(a_t|s_t;\theta)) = -\sum_{i=1}^{M} P(a_i|s_t;\theta) \log P(a_i|s_t;\theta), \tag{4.8}$$

where $\pi(a_t|s_t;\theta) = [P(a_1|s_t;\theta), \ldots, P(a_M|s_t;\theta)]$ for an action set of M actions. Note that the probabilities defined by π now are parameterized by θ corresponding to the policy-specific weights in the CNN model. As evident by (4.8), the entropy is maximized for policies where the probability of any given action comes from a uniform distribution where $P(a_i|s_t;\theta) = 1/M$, i.e. when the policies display the least deterministic behaviour. Hence, the entropy $H(\pi(a_t|s_t;\theta))$ can be used as a tool to motivate the agent to steer clear of less non-deterministic policies if it is added to (4.6). This procedure is referred to as *entropy regularization* [21]. With entropy, the objective function for the actor is now represented by

$$f(\theta, \theta_v) = \log \pi(a_t|s_t;\theta)(R_t - V(s_t;\theta_v)) + \beta H(\pi(a_t|s_t;\theta)), \tag{4.9}$$

where β is a coefficient dictating the magnitude of regularization. As less non-deterministic policies represent distributions with a heavy bias toward a specific subset of actions in $\mathcal{A}$, entropy regularization makes it possible for the agent to sample actions that may be necessary to leave local optimas of the objective function in order to progress training.

4.4.2 Numerical Optimization and Back-propagation

Generally, the objective function is seen as a loss function. Intuitively, this means that the negative value of (4.9) should be minimized. This leads to the loss function formulated by

$$L(\theta, \theta_v) = -\log \pi(a_t|s_t;\theta)(R_t - V(s_t;\theta_v)) - \beta H(\pi(a_t|s_t;\theta)), \tag{4.10}$$

which corresponds to the actor represented by the policy. The critic follows a separate function based on the L_2-loss of the advantage function according to

$$L_v(\theta_v) = (R_t - V(s_t;\theta_v))^2, \tag{4.11}$$

which also is subject to minimization as it ought to converge to zero during the course of training. From Sections 4.3 and 4.4, it is evident that the optimization problem of minimizing (4.10) and (4.11) with respect to the model parameters θ and θ_v is non-trivial. The resulting formulation of the minimization problem after incorporation of (4.11) follows

$$\begin{aligned} \text{minimize} \quad & \tilde{L}(\theta, \theta_v) = L(\theta, \theta_v) + L_v(\theta_v) \\ \text{subject to} \quad & \sum_i \pi(a_i|s;\theta) = 1 \\ & \pi(a_i|s;\theta) \in (0,1) \quad \forall a_i \in \mathcal{A} \\ & s \in \mathcal{S} \\ & \tilde{L} \in \mathbb{R} \end{aligned} \tag{4.12}$$

which is constrained merely by the need to keep the loss functions finite. Given previous motivations, the optimization problem is solvable numerically with stochastic gradient descent based methods by leveraging the fact that the neural network is differentiable. The gradients of $\tilde{L}(\theta, \theta_v)$ with respect to the network parameters $\boldsymbol{\theta} = [\theta, \theta_v]$ carries the formulation

$$\begin{aligned} \nabla \tilde{L}(\boldsymbol{\theta}) &= \nabla L(\theta, \theta_v) + \nabla L_v(\theta_v) \\ &= \left(\frac{\partial L(\theta, \theta_v)}{\partial \theta}, \frac{\partial L(\theta, \theta_v)}{\partial \theta_v} + \frac{\partial L_v(\theta_v)}{\partial \theta_v} \right). \end{aligned} \tag{4.13}$$

Naturally gradient descent based minimization is performed by following the negative gradient direction which corresponds to the steepest descent possible from any point in $\tilde{L}(\theta, \theta_v)$ with some step-size α, otherwise known as *learning rate* in machine learning. Given the loss function $\tilde{L}(\theta, \theta_v)$, the parameters θ and θ_v are updated according to

$$\begin{aligned} \theta_{t+1} &\leftarrow \theta_t - \alpha \frac{\partial L}{\partial \theta}, \\ \theta_{v,t+1} &\leftarrow \theta_{v,t} - \alpha \left(\frac{\partial L}{\partial \theta_v} + \frac{\partial L_v}{\partial \theta_v} \right). \end{aligned} \tag{4.14}$$

An example illustration of gradient descent minimization of a paraboloid $\tilde{L}(\boldsymbol{\theta})$ for $\boldsymbol{\theta} = [\theta, \theta_v]$ can be seen in Figure 4.2.

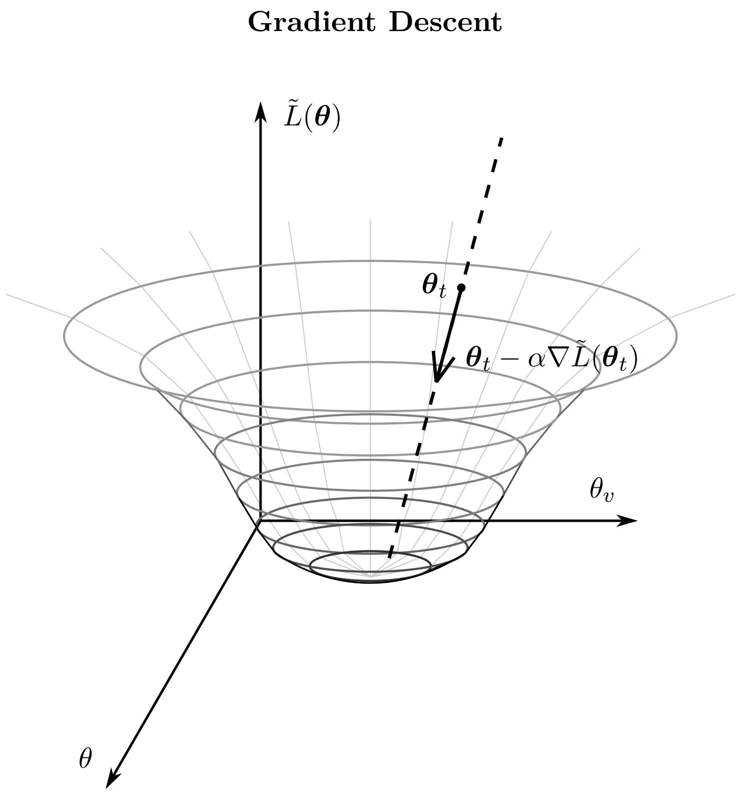

Gradient Descent
$\tilde{L}(\boldsymbol{\theta})$
$\boldsymbol{\theta}_t$
$\boldsymbol{\theta}_t - \alpha\nabla\tilde{L}(\boldsymbol{\theta}_t)$
θ_v
θ

Figure 4.2: Illustration of the gradient descent procedure for a paraboloid loss function $\tilde{L}(\boldsymbol{\theta})$. By iteratively performing gradient descent steps, the procedure leads to the location of the loss function optima.

When updating θ and θ_v using the gradients of the loss function, these parameters are changed with what is referred to as error *back-propagation*. When using gradient descent, the learning rate is key in order to acquire a minimal value of the loss function. If the magnitude of descent is too large, there is a risk of never finding an optima as the descent step would overshoot its location. Further, if the learning rate is too small, there is a possibility of prematurely finding local optimal values even if the loss function could be further minimized or maximized in either another local optima or even a global one.

4.4.3 RMSProp

Modern methods based on stochastic gradient descent often use an adaptive learning rate which changes following the properties of the gradients themselves such as their magnitude. In the original A3C implementation, a modified version of the RMSProp optimizer proposed by Tieleman et al. is used to limit the stepsizes of the descent [22]. The core mechanic of RMSProp is to use a moving average of the loss function gradients over a span of forward passes in the network as a tool for yielding learning rates based on the immediate history of the loss function. This quality makes the method robust and agile when faced with heavily varying loss function gradients. An incremental step in the RMSProp method can be formulated as

$$\begin{aligned} \mathbf{g}_t &= \nabla_{\boldsymbol{\theta}_t} \tilde{L}(\boldsymbol{\theta}_t), && (4.15) \\ \mathbf{r}_t &= \alpha_r \mathbf{r} + (1-\alpha_r)\mathbf{g} \odot \mathbf{g}, && (4.16) \\ \boldsymbol{\theta}_{t+1} &\leftarrow \boldsymbol{\theta}_t - \alpha \frac{\mathbf{g}}{\sqrt{\mathbf{r}+\epsilon}}, && (4.17) \end{aligned}$$

where α_r is the decay rate of the update, α the learning rate and ϵ a small constant inhibiting numerical instability for division by $\mathbf{r}$-elements close to zero. The gradients are represented by $\mathbf{g}$ and their squared counterpart by $\mathbf{r}$ computed as the element-wise Hadamard product.

4.5 Algorithmic Formulation

The algorithmic formulation of A3C for each parallel learner agent, summarizing above sections, is presented in Algorithm 1. Note that the entropy regularization is used in practice but omitted according to the algorithmic formulation in the original article [1].

Algorithm 1 Asynchronous Advantage Actor-Critic [1]

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

Schematically, each learner agent has a local copy of a global neural network that is synchronized before each episode of at most t_{max} steps. Note that t_{max} is merely the number of forward passes in the network allowed for each agent, yielding policies from which actions are sampled for the environment interactions and not the timestep where the algorithm terminates training as per Section 2.1. In addition, each agent maintains its own instance of the problem environment. The agents train the global neural network using RMSProp in parallel via the lock-free Hogwild paradigm [23].

Chapter 5

Improvements

The investigated improvements are discussed in separate sections including the Adam optimization method and layer normalization in combination with Long Short-Term Memory (LSTM).

5.1 Adam Optimizer

The Adam optimizer is a gradient-based optimization model for stochastic objective functions utilizing the estimates of running averages of the first two orders of moment, which is also the origin of its name - adaptive moment estimation. Similar to RMSProp, Adam works well for on-line solution methods where the reinforcement learning algorithm processes input data in a step-by-step manner as opposed to batched, off-line learning. A prominent difference between RMSProp and Adam is the use of a momentum on the rescaled gradients of the loss function $L(\theta)$ in RMSProp compared to the moment estimates of Adam [2]. Being proposed as a possible improvement to A3C in its original implementation, Adam is worthy of further examination [1]. As in the case of RMSProp, Adam can be applied either in a shared setting or with moment estimates separated over each training thread in the A3C algorithm. A step in the procedure of computing the gradient updates in the Adam method is presented in Algorithm 2.

Algorithm 2 Algorithmic formulation of the Adam method for a stochastic objective function $L(\theta)$ [2].

```
Require: Learning rate: $\alpha_{lr}$
Require: Exponential Decay Rates: $\beta_1, \beta_2 \in [0, 1)$
Require: Initial Model Parameter set $\theta_0$
 1: $m_0 \leftarrow 0$
 2: $v_0 \leftarrow 0$
 3: $t \leftarrow 0$
 4: while $\theta_t$ not converged do
 5:     $t \leftarrow t + 1$
 6:     $g_t \leftarrow \nabla_\theta L(\theta_{t-1})$
 7:     $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
 8:     $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
 9:     $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$
10:     $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$
11:     $\theta_t \leftarrow \theta_{t-1} - \alpha_{lr} \frac{\hat{m}_t}{\epsilon_a + \sqrt{\hat{v}_t}}$
12: end while
13: return $\theta_t$
```

In Algorithm 2, v_t and m_t are the first and second order moment estimates of the gradients prior to bias correction. As the effective step in the parameter space of the reinforcement learning model performed with the gradient moment estimates are bounded by the learning rate α_{lr}, the Adam method is insensitive to sudden changes in the magnitude of the gradients of loss function which limits the need for extensive hyper parameter searches for viable learning rates [2].

5.2 Addition of Memory and Layer Normalization

In problems where there may be temporal dependencies between a received reward at timestep t and information from previous inputs $s_{t-1}, s_{t-2}, \ldots,$ there is an incentive to sustain some knowledge of input history. It is wasteful to discard information that may be useful at a later point in time and even destructive for the training process if this information proves to be vital. The biological analogy to this experience persistence is the short- or long-term memory which enables us to avoid relearning things previously known. As mentioned in Section 4.3, the input observations are represented as stacks of four images in order to account for the motion of in-game objects. By using some sort of memory mechanism, the spatial trajectory of a given in-game object should be learnable without the need to feed the agent multiple consecutive frames. One approach of sustaining the context of a problem in a sequence of time is by using recurrent neural networks (RNN). In an RNN architecture, the hidden representation h_t of a layer at time t for some input s_t is kept for consecutive timesteps and fed back into the network with the use of a special set of weights corresponding to the connections between h_t and h_{t-1} for the same layer. By looking one step further back in time the same applies for h_{t-1} and

h_{t-2}. It is this recursive nature of an RNN that gives it its name. Due to conflicting naming, the input states s used for the observations in the Atari environment will temporarily be renamed x in the following section to avoid confusion, i.e. $s_t = x_t$.

5.2.1 Long Short-Term Memory

In a deep learning setting, a well used RNN architecture is the *long short-term memory* (LSTM) model which stems from a set of RNN models that are referred to as gated [11,24]. For gated RNN, the mechanism behind the information persistence in the network is governed by gates that actively decides what should happen to previous information at future timesteps. In LSTM there are three gate types governing input, output as well as the act of forgetting information of a certain age. Each of these controls whether the long short-term memory is read from, written to, or reset during training [25]. Key to the structure of the LSTM cell is the state unit $s_i^{(t)}$. The state unit contains the information of the LSTM cell and by use of the internal cell gates, this information can be changed. The activation of each gate is represented by the sigmoid function σ, yielding smooth gate outputs in the range of $(0, 1)$ which by extension sustains the differentiability of the entire neural network model containing the LSTM cell [25]. The forget gate unit formulated by

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right), \tag{5.1}$$

controls the degree of kept information in state unit $s_i^{(t)}$ given its gate-specific input weights U^f, recurrent weights W^f and biases b^f. An output of 0 from the sigmoid function corresponds to completely forgetting said information and 1 the act of saving it entirely for the next consecutive timestep. The state unit of the LSTM cell is updated by

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right), \tag{5.2}$$

where the product $f_i^{(t)} s_i^{(t-1)}$ corresponds to the degree of forgotten information from the previous time step $t-1$. The amount of LSTM cell input from any prior network layer and the LSTM cell output is governed by the external input gate

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right), \tag{5.3}$$

as well as the output gate

$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}\right), \tag{5.4}$$

which essentially limit the external interaction with the cell. These gates too have their respective parameter sets U^g, W^g, b^g, U^o, W^o and b^o in accordance with the forget gate. The final LSTM cell output comes from the hyperbolic tangent function

$$h_i^{(t)} = \tanh\left(s_i^{(t)}\right) q_i^{(t)}, \tag{5.5}$$

which represents the activation function of the LSTM layer. The algorithmic formulation of a forward pass in the LSTM cell is summarized in Algorithm 3.

Algorithm 3 Algorithmic formulation of the LSTM forward pass

```
Require: LSTM cell state: $c$
Require: LSTM hidden representation: $h$
Require: Number of units in LSTM cell: $N$
Require: Minibatch size: $M$
Require: Input: $x_0, x_1, \dots, x_{\tau-1}, x_\tau$
Require: Trainable parameters: $W_x \in \mathbb{R}^{M\times 4N}$
Require: Traininable parameters: $W_h \in \mathbb{R}^{N\times 4N}$
 1: for $0 \le i \le \tau$ do
 2:     if $x_i$ terminal then
 3:         $c \leftarrow 0$
 4:         $h \leftarrow 0$
 5:     end if
 6:     $z \leftarrow x_i^\mathsf{T} W_x + h^\mathsf{T} W_h + b$
 7:     $\begin{bmatrix} z_i & z_f & z_o & z_u \end{bmatrix}^\mathsf{T} \leftarrow z$
 8:     $z_i \leftarrow \sigma(z_i)$
 9:     $z_f \leftarrow \sigma(z_f)$
10:     $z_o \leftarrow \sigma(z_o)$
11:     $z_u \leftarrow \tanh(z_u)$
12:     $c \leftarrow z_f c + z_i z_u$
13:     $h \leftarrow z_o \tanh(c)$
14:     $y_i \leftarrow h$
15: end for
16: return $y_0, y_1, \dots, y_{\tau-1}, y_\tau$
```

Introducing such an LSTM cell in the neural network model, time delays between interaction and reward can be compensated which by extension lets the agent base its actions on previous events. Specifically, in the original A3C model architecture the third and fully connected layer is substituted with an LSTM cell as displayed in Figure 5.1.

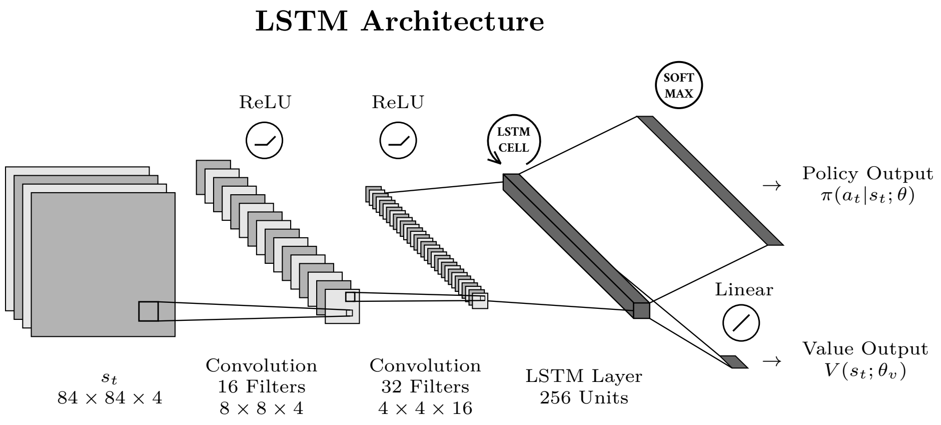

LSTM Architecture
ReLU
ReLU
LSTM CELL
SOFT MAX
Policy Output $\pi(a_t|s_t;\theta)$
Linear
Value Output $V(s_t;\theta_v)$
s_t $84 \times 84 \times 4$
Convolution 16 Filters $8 \times 8 \times 4$
Convolution 32 Filters $4 \times 4 \times 16$
LSTM Layer 256 Units

Figure 5.1: Original A3C model architecture similar to Figure 4.1 where the third fully connected layer of 256 units is replaced with an LSTM cell. Note that the input in this illustration is represented as a stack of four greyscale images which can be replaced with a single image to limit data pass-through.

5.2.2 Layer Normalization

A state-of-the-art reinforcement learning model using LSTM is an expensive architecture to train as evident by the abundance of internal parameters shown in Section 5.2.1. Even if A3C by itself is considered efficient via its use of multiple training threads, it still requires about 24 hours worth of training in its original formulation [1]. In recent time, *batch normalization* has been one popular method for improving training times for training batched, offline supervised learning models by normalizing the input over a batch of training samples for a single neuron. However, models using batch normalization tend to be less sensitive to high learning rates and weight initialization [26]. With A3C being an online method for solving reinforcement learning problems, batch normalization is not readily applicable especially given the temporal complexities introduced by a potential LSTM layer. By instead normalizing the input via a single training sample over all the neurons in a given layer, it is possible to disregard the need for batched samples as discovered by Ba et. al. [3]. In practice, the normalization statistics in the form of the mean and variance for layer i over all of its H number of neurons are computed with

$$\begin{aligned} \mu_i &= \frac{1}{H}\sum_{j=1}^{H} a_{i,j}, \\ \sigma_i &= \sqrt{\frac{1}{H}\sum_{j=1}^{H}(a_{i,j}-\mu_i)^2}, \end{aligned} \tag{5.6}$$

where $a_{i,j}$ is the activation in the hidden layer $\mathbf{h}_i$ corresponding to neuron j. Using (5.6), the input to layer i is normalized with

$$\hat{\mathbf{h}}_i = g\left[\frac{\mathbf{g}_{i-1}}{\sigma_{i-1}} \odot \left(\mathbf{W}_{i-1}^T \mathbf{h}_{i-1} - \mu_{i-1}\right) + \mathbf{b}_{i-1}\right], \tag{5.7}$$

again where $\odot$ constitutes the Hadamard product. Here, the gain $\mathbf{g}_i$ shares the dimensionality of $\mathbf{h}_i$ and scales the normalized input for each neuron in the layer respectively. The bias $\mathbf{b}_i$ of layer i follows the explanation in Section 2.4. Both $\mathbf{g}_i$ and $\mathbf{b}_i$ are updated during the course of training. Normalization for an LSTM layer is, in comparison to the batch normalization case, trivial to implement. Instead of using the normalization statistics from the output the previous layer $i-1$ in the network, the corresponding ones are used but for the temporally unrolled, hidden LSTM state at time $t-1$. Hence, by indexing μ, σ and $\mathbf{h}$ with t the RNN formulation of layer normalization follows

$$\hat{\mathbf{h}}_t = g\left[\frac{\mathbf{g}_t}{\sigma_t} \odot \left(\mathbf{W}_t^T \mathbf{h}_t - \mu_t\right) + \mathbf{b}_t\right]. \tag{5.8}$$

The layer normalization is applied to the recursive hidden layer in the LSTM model displayed in Figure 5.1.

Chapter 6

Implementation

6.1 Network Initialization

The network initialization procedure used in the A3C implementation of this study is based on a modification of the commonly used Xavier initialization which leverages the structure of the network as proposed by Xavier Glorot and Yoshua Bengio [27]. In the original formulation of the initialization procedure, the weights $\mathbf{W}_j$ of a given layer are sampled from a uniform distribution according to

$$w_i \sim U\left(-\sqrt{\frac{6}{n_j + n_{j+1}}}, \sqrt{\frac{6}{n_j + n_{j+1}}}\right), \quad \forall w_i \in \mathbf{W}_j \tag{6.1}$$

where n_j and n_{j+1} are the number of incoming connections from previous layer and outgoing to the next. This method alleviates the need to choose a initialization distribution on a case-by-case basis. With the motivation that the original Xavier initialization does not take nonlinear network activation functions into consideration, He et al. proposed a modification to (6.1) by sampling the elements of $\mathbf{W}$ for a given layer j from a normal distribution based on the second moment statistics of the preceding weights when using the ReLU activation function [28]. Hence, the initial network weights w_i for layer j are sampled according to

$$w_i \sim N\left(0, \sqrt{\frac{2}{n_j}}\right), \quad \forall w_i \in \mathbf{W}_j \tag{6.2}$$

for n_j number of weights in the previous layer. The importance behind a well-initialized network is the impact it has on data fed forward through the network. If the layer weights are too large the neural activity in the network will become saturated which, depending on the activation functions used in the network, may lead to small enough gradients to essentially inhibit training or make it infeasibly slow [29]. As the bias parameters merely translates the output of a given layer, it has less of an immediate impact and may be initialized to zero.

6.2 Experimental Setup

The baseline experiments are designed to reproduce the results acheived by Mnih et al. in the original paper [1]. For RMSProp, $\alpha_r = 0.99$ and $\epsilon = 0.1$. From correspondence with the authors of the original A3C article, the optimal initial learning rate for each learner agent proves to be $7 \cdot 10^{-4}$ which is linearly annealed to zero during the course of training [30]. The L_2-norm of accumulated gradients before back-propagation is clipped to 40. The reward received from the Atari emulator is clipped to $r_t \in [-1, 1]$ [1,13]. The magnitude of the entropy regularization in the objective function is scaled with $\beta = 0.01$. Each forward pass rollout is limited to at most $t_{max} = 5$ or the number of steps until the termination of the current episode as shown in Algorithm 1. The number of parallel learner agents used amounts to 16. In the Adam experiments, every hyper parameter is reused from the baseline experiments. Instead of linearly annealing the learning rate, it is kept constant. The RMSProp optimization method is replaced with the Adam optimizer using the parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$ [2]. The Adam-specific ϵ_a-parameter is set to 10^{-8}. In the final experiment set using the layer normalized LSTM cell, the RMSProp optimizer is reinstated with the same hyper parameters as in the baseline experiments. In this case, two observation types are used corresponding to a stack of four frames as well as a single frame. All of these experiments are performed for Breakout, Pong and Space Invaders respectively.

6.3 Run Statistics and GPU Based Architecture

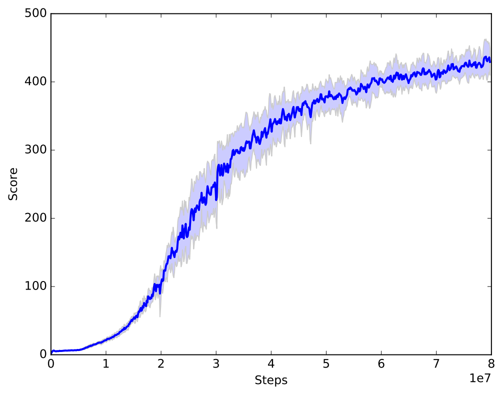
In the original A3C paper, Google DeepMind performs 50 runs for each Atari game. From these results, the average of the 5 best performing models are presented [1]. In this study every postulated experiment is run 5 times and the result from each experiment is represented by the mean over these 5 runs together with their corresponding standard deviations. These results should be a fair representation of the actual performance of the A3C algorithm without cherrypicking the best trained models based on their internal ranking with respect to their final scores. This A3C implementation was developed at SEED by myself and Henrik Holst using the TensorFlow framework. Our implementation partitions the algorithm into a server-client topology where a server holds a central copy of the neural network model which is then fed data from a set of clients corresponding to the learner agents complying with the asynchronous multi threaded architecture described in the original article [1]. By keeping the gradient back-propagation operations on the server, it is possible to accelerate them with a GPU leveraging the NVIDIA CUDA backend of TensorFlow. All clients are run on Amazon's AWS c4.4xlarge instances which as of October 2016 are configured with Intel's Xeon E5-2666 v3 processors of 16 CPU cores and 30 GiB of RAM. The server process is run on the p2.xlarge instance type with an NVIDIA K80 GPU, 61 GiB of RAM and 4 CPU cores.

Chapter 7

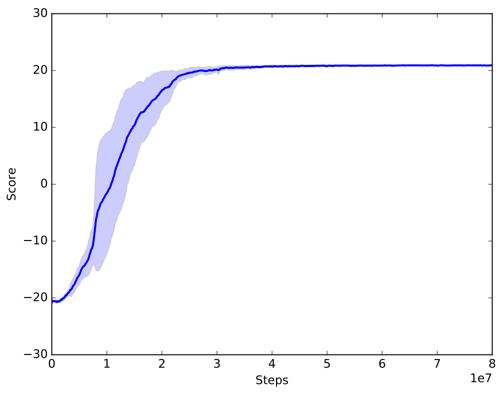
Results

Using the hyper parameters suggested by the original authors of the A3C paper as presented in Section 6.2, the performance of A3C is gauged for Breakout, Pong and Space Invaders as shown in Figure 7.1. Replacing RMSProp with the Adam optimization method yields the results displayed in Figure 7.2. Using the environment observation shape of $s_t \in \mathbb{R}^{84\times84\times4}$ as for the other runs and reverting the optimizer to RMSProp, the fully connected layer of the original model is replaced with a layer normalized LSTM cell as described in Section 5.2. The results from these runs are illustrated in Figure 7.3. Finally, the results from leveraging the LSTM cell when the observations are reduced to single images $s_t \in \mathbb{R}^{84\times84}$, denoted by No FS (No Framestack), are illustrated in Figure 7.4. From now on, the collection of learner agents is simply referred to as the A3C agent.

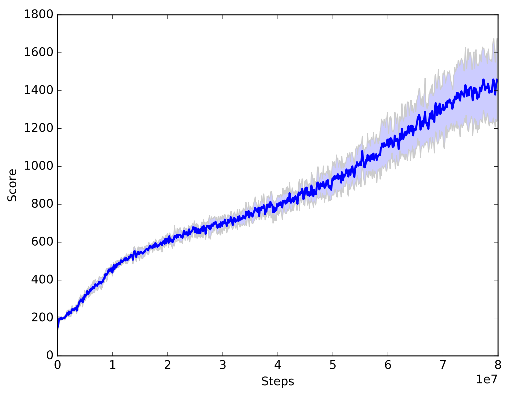
Baseline



(a) Breakout



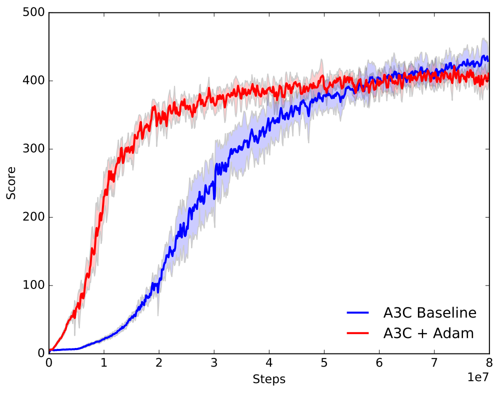
(b) Pong



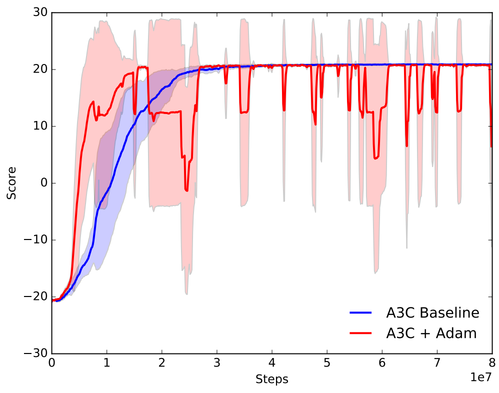
(c) Space Invaders

Figure 7.1: In-game scores for the A3C agent plotted against the number of training steps in Breakout, Pong and Space Invaders using the baseline setup.

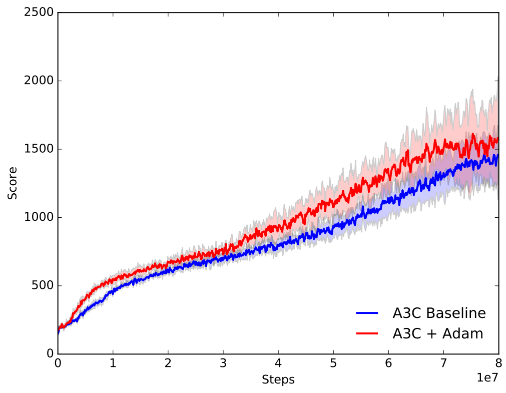
Adam



(a) Breakout



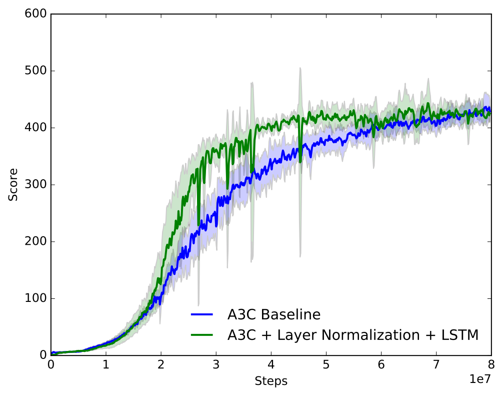
(b) Pong



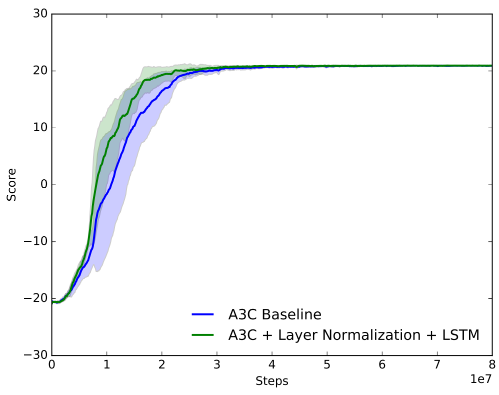
(c) Space Invaders

Figure 7.2: Scores plotted against the number of training steps with the Adam optimization method in Breakout, Pong and Space Invaders.

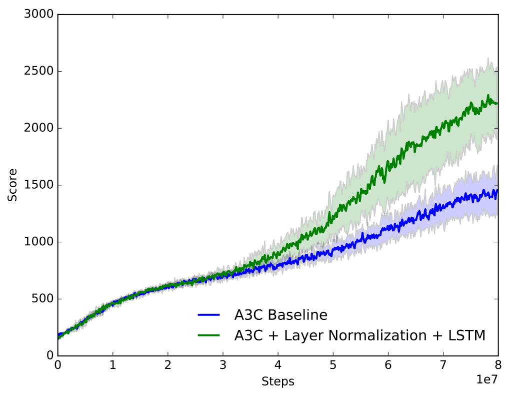
LSTM + LN



(a) Breakout



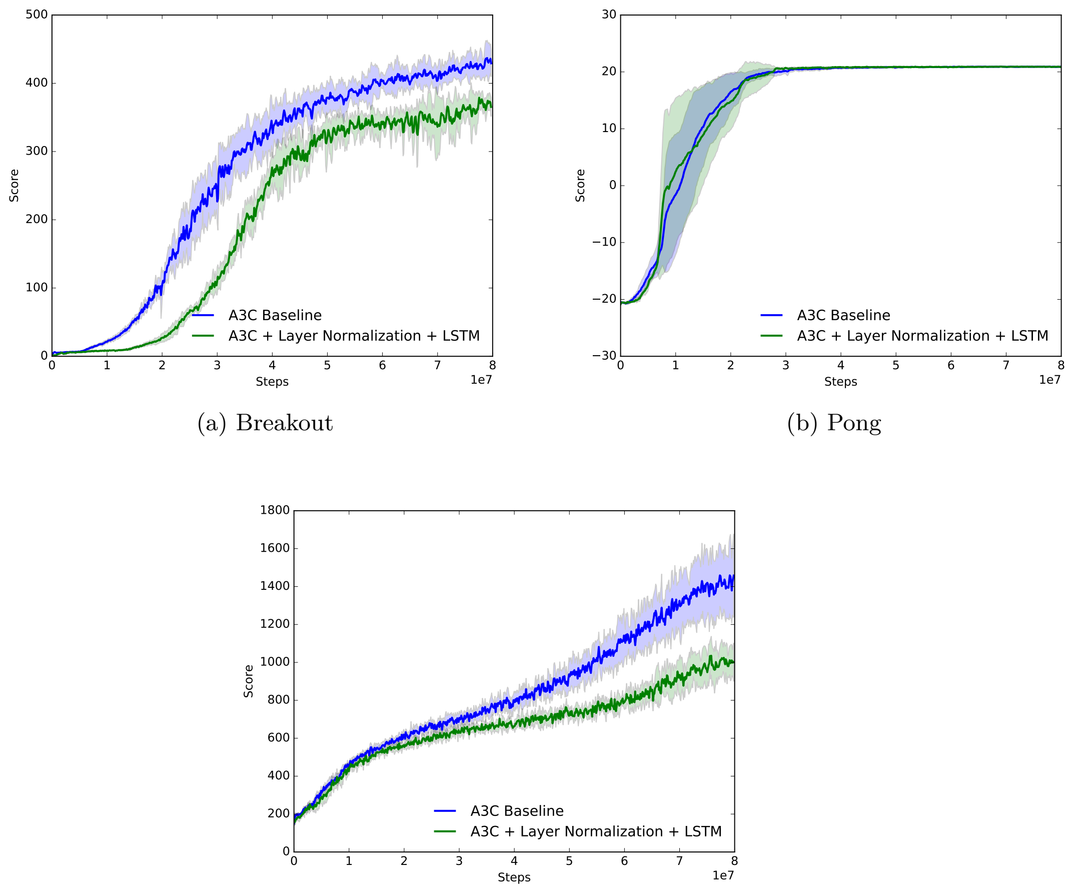
(b) Pong



(c) Space Invaders

Figure 7.3: Scores for the A3C agent using the LSTM + LN model architecture for Breakout, Pong and Space Invaders using an observation image stack of 4.

LSTM + LN + No FS



(a) Breakout

(b) Pong

(c) Space Invaders

Figure 7.4: Scores plotted against iteration steps for the LSTM + LN model. In this experiment each observation fed to the network is of only one image eliminating the framestack, hence forcing the A3C agent to rely on the LSTM cell to sustain a temporal awareness of the environment.

7.1 Training Time and Speed-up

A metric gauging the computational complexity of each experiment is the one of wall-clock training time. The number of hours needed to train the A3C agent for 80M steps is illustrated in Table 7.1 for each experiment.

Training Time

	Breakout	Pong	Space Invaders
Baseline (h)	8.7	8.2	8.0
Adam (h)	8.6	8.1	8.1
LSTM + LN (h)	15.8	16.0	16.4
LSTM + LN + No FS (h)	14.5	14.6	14.5

Table 7.1: Number of hours needed to train the A3C agent for 80M iterations for each experiment. The last row corresponds to the agent using LSTM and layer normalization but without the framestack of 4 images.

From the score graphs it is clear that each and every applied improvement has an impact on the mean score trajectory for each game. In some cases, the mean score after finalized training is even slightly worse than that of the corresponding baseline runs in exchange for much improved, initial training performance. By setting a cutoff level in the baseline runs when a game is considered solved, it could be used to find the number of steps needed to solve the games for the rest of the experiments. The level is chosen to represent 75% of the final baseline scores. The number of steps needed to reach this level for each improvement is shown in Table 7.2. This score is henceforth referred to as $b_{0.75}$ and the utility of this metric is further discussed in Section 8.1.

Training Steps

	Breakout	Pong	Space Invaders
Baseline (10^6)	36.7	19.4	55.3
Adam (10^6)	16.2	11.4	45.5
LSTM + LN (10^6)	26.4	15.4	44.3
LSTM + LN + No FS (10^6)	47.6	20.9	–

Table 7.2: Number of updates needed for the agent to reach the mean score level $b_{0.75}$ corresponding to 75% of the final baseline score of each game in the end of the training session of 80M steps. The baseline column displays the number of steps needed to reach this level without any of the improvements. For Pong, Breakout and Space Invaders the actual $b_{0.75}$ scores are 322.7, 15.7 and 1065.8 respectively.

Chapter 8

Conclusion and Discussion

8.1 Interpreting the Results

Comparison to DeepMind's Results

Given the motivation of using the mean score over multiple runs as mentioned in Section 6.3, there is still value in comparing the baseline results of this study with those produced by Google DeepMind in order to verify that the A3C implementation is working. The first row of Table 8.1 contains the final scores for Google DeepMind's A3C agent trained for 80M steps on Breakout, Pong and Space Invaders [1]. The final scores from the experiments of this study are represented in the remaining rows.

Result Summary - Final Mean Scores

	Breakout	Pong	Space Invaders
DeepMind's Results	551.6	11.6	2214.7
Baseline	430.2	20.9	1421.1
Adam	405.1	20.8	1552.0
LSTM + LN	422.7	20.9	2223.9
LSTM + LN + No FS	370.6	20.9	1000.6

Table 8.1: Final mean score for each game and proposed improvement after 80M steps of training including DeepMind's baseline results represented by the first row.

Using the default hyper parameters provided in the A3C article as well as the information disclosed by the authors, the baseline results differ significantly with those obtained by DeepMind. For Pong, the end mean score is almost a factor of two larger than that of the original paper. It may be so that the network initialization differs between the two implementations resulting in greatly different mean score trajectories during training, which is point of interest not described in the original article [1].

Baseline vs. Adam

By exchanging the RMSProp optimizer with Adam, the A3C agent reaches higher or equivalent scores to that of the baseline at much fewer steps. For Breakout, the final score is surprisingly 25.1 points less than that of the baseline run as seen in Table 8.1 with 405.1 points compared to 430.2. However, evident from Figure 7.2a, the actual training of the model is much more efficient which also is reflected in the number of steps needed to reach the $b_{0.75}$ score level in Table 7.2. Using Adam, it takes less than half the number of training steps until $b_{0.75}$ is reached in the baseline run - i.e. 16.2M steps compared to 36.7M. The main interest in this experimentation group is the instability in the case of Pong as seen in Figure 7.2b as the A3C agent occasionally drops down to a score of -20 which is almost the lowest score possible. The root of this instability may be the sparse reward signal of the game as discussed in Section 8.1. The $b_{0.75}$ level is reached faster than in the baseline scenario, needing only 11.5M steps instead of 19.4M with a final score of 20.8. For Space Invaders, Adam seems to have little positive effect only off-setting the final score of merely 130.9 points, leading to a final score of 1552.0 compared to 1421.1 which also is well within the standard deviation. The $b_{0.75}$ level is reached in 45.5M steps compared to 55.3M.

Baseline vs. LSTM + LN

Using LSTM with layer normalization and an observation stack of four images in combination with the RMSProp optimizer, the differences to the baseline runs are less severe than in the Adam experiment set. In Breakout, the agent reaches the $b_{0.75}$ level at 26.4M steps which is slower than in the Adam case. The final mean score is better than with the Adam optimizer but still slightly worse than the final baseline score. The performance in Pong is relatively unchanged with a decrease in steps needed to reach $b_{0.75}$ to 15.4, compared to the baseline with a final score of 20.9. The stability of the model is greatly increased using LSTM + LN which may be attributed to the increased ability of the agent to correlate its actions with the high latency rewards given the nature of the game. The main area of interest in this experiment group is the results from the Space Invaders runs. The final mean score is significantly higher at 2223.9 compared to 1421.1 in the baseline, which is also improved in comparison with the Adam results. The $b_{0.75}$ level is reached at 44.3 which is marginally better than Adam.

Baseline vs. LSTM + LN + No FS

When decreasing the observation image stack to one grayscale image, the performance in all three games is negatively impacted. The A3C agent drops down to a final mean score of 370.6 in Breakout and increases the number of steps needed to reach $b_{0.75}$ to 47.6M compared to 36.7M. The corresponding results for Pong shows the same behaviour with 20.9M steps. The final score remains at 20.9. The performance in Space Invaders is the most surprising with a decrease of final score to 1000.6. As a result, the agent fails to reach the $b_{0.75}$ level before the 80M mark when the training is terminated.

Speed-up and Data Efficiency

The results in Table 7.1 clearly shows the impact of the modifications to the A3C algorithm. The only improvement not imposing an increase in computational complexity is the change of RMSProp to Adam. It is the introduction of the LSTM cell that constitutes the largest offset in wall-clock training time. For all games, the number of hours needed to train the A3C agent for 80M steps increase by approximately a factor of two which may be attributed to the increased number of parameters in the model. Out of the three games, the agent's performance is only improved in Space Invaders. However, this improvement is the largest out of all experiments. As discussed in Section 5.2, the addition of the LSTM memory should allow for successful training even with single-frame observations as the magnitude of data fed through the network is decreased by a factor of four. Being possibly bandwidth bound in regards of GPU, network communication or CPU utilization, this may be a viable candidate of yielding trained agents still performing according to the baseline but without the observation size drawback. However, when reviewing the results, the final scores decrease in both Breakout and Space Invaders and the number of hours needed to reach 80M training steps is lessened by merely 1.5 hours resulting in a mean training time of 14.5 hours compared to the baseline mean of 8.3 which also coincides with the training time when using the Adam optimizer. This minute difference implies that the use of LSTM + LN with single-image observations provides little value when training an agent on these games even considering the reduction in data through-put.

8.2 Summary

During the experimentation in this study, it is evident that Pong is the most difficult game for the A3C agent to play in both a successful but also consistent manner. A reason behind this could be amounted to the specific score signature of the game. In the other two games the reward stream is somewhat consistent and more frequent. In Pong, the agent has to not only stop the ball from leaving the play area but also hit the ball such that it scores against the built-in rudimentary in-game opponent which may take hundreds of in-game timesteps after the agent performs the necessary action to do so. In the other two games, the agent receives rewards from simply breaking the bricks in Breakout or destroying single alien spaceships in Space Invaders. As these reward structures are more dense, the agent is fed with a more accurate report on the current play which stimulates the training of the network model. This is especially important during the initial phase of training as the weights of the convolutional layers are updated. This is when the agent learns the correlations between the interaction with the environment and the resulting change in the visual state of the in-game images fed to the network. A3C performs best with use of the Adam optimizer, resulting in faster training for all games. The LSTM + LN addition provides more stability to the model with the side-effect of decreasing the initial training performance compared to the Adam case. Using single-image observations in conjunction with LSTM + LN decreases training time in hours compared to using observation stacks of four frames. This modification decreases the performance in both Breakout and Space Invaders. As implied, this particular improvement requires some more investigation.

8.3 Future Work

Being an on-policy asynchronous actor-critic model, the addition of a working experience replay memory akin to that used in DQN is something that is a logical step of improvement. This is also proposed by the original authors [1, 13]. Not only would such a memory improve the data efficiency of the A3C algorithm even further, but also allow for the possibility to complement the reinforcement learning setting with a supervised learning entrypoint. This entrypoint could be used to bootstrap the initial training of the convolutional network using prerecorded gameplay data from human players in an imitation learning setting. Increasing the network depth and complexity by introducing more convolutional layers should also allow for better exploration of the latent variable space within the states of the reinforcement learning environment. As the Atari 2600 games investigated in this study are fully observable, the model described in the original paper may not be sufficient for more advanced environments such as 3D games where observations may be partial. The convolutional kernel matrices in the original network architecture could be exchanged with smaller filters which would then allow for the more efficient Winograd convolution operation.

Bibliography

[1] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[2] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[3] Jamie Ryan Kiros Jimmy Lei Ba and Geoffrey E. Hinton. Layer normalization. https://arxiv.org/abs/1607.06450, 2016.

[4] Wolfram Schultz, Peter Dayan, and P. Read Montague. A neural substrate of prediction and reward. *Science*, 275(5306):1593–1599, 1997.

[5] Nathaniel D Daw and Philippe N Tobler. Value learning through reinforcement: The basics of dopamine and reinforcement learning. pages 283–298, 01 2014.

[6] Kent C Berridge and Terry E Robinson. What is the role of dopamine in reward: hedonic impact, reward learning, or incentive salience? *Brain Research Reviews*, 28(3):309 – 369, 1998.

[7] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[8] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

[9] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016. Article.

[10] John Tromp and Gunnar Farnebäck. *Combinatorics of Go*, pages 84–99. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[12] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*. 2013.

[15] Marc G. Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Rémi Munos. Unifying count-based exploration and intrinsic motivation. *CoRR*, abs/1606.01868, 2016.

[16] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 06 2013.

[17] Nick Montfort and Ian Bogost. *Racing the beam : the Atari Video computer system*. Cambridge, Mass. : MIT Press, Cambridge, Mass., 2009. Includes bibliographical references (p. [159]-167) and index.

[18] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively parallel methods for deep reinforcement learning. *CoRR*, abs/1507.04296, 2015.

[19] Verena Heidrich-Meisner, Martin Lauer, Christian Igel, and Martin Riedmiller. Reinforcement learning in a nutshell. pages 277–288, 01 2007.

[20] Nicolas Heess, David Silver, and Yee Whye Teh. Actor-critic reinforcement learning with energy-based policies. In Marc Peter Deisenroth, Csaba Szepesvári, and Jan Peters, editors, *EWRL*, volume 24 of *JMLR Proceedings*, pages 43–58. JMLR.org, 2012.

[21] Ronald J. Williams and Jing Peng. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268, 1991.

[22] T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[23] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc., 2011.

[24] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[25] Andrej Karpathy, Justin Johnson, and Fei-Fei Li. Visualizing and understanding recurrent networks. *CoRR*, abs/1506.02078, 2015.

[26] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[27] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[29] Russell D. Reed and Robert J. Marks. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks.* MIT Press, Cambridge, MA, USA, 1998.

[30] Github User Muupan's Correspondence with Dr. V. Mnih, 2016. Git Wiki: https://github.com/muupan/async-rl/wiki.

TRITA -MAT-E 2017:81
ISRN -KTH/MAT/E--17/81--SE