Application of convolutional neural networks to RL control problems

by Marek Romanowicz (T)

Fourth-year undergraduate project in
Group F, 2014-2015

I hereby declare that, except where specifically indicated, the work submitted herein is my own original work.

Signed: _____ Date: _____

Technical Abstract

Application of convolutional neural networks to RL control problems

by

Marek Romanowicz (T)

Fourth-year undergraduate project in Group F, 2014-2015

Objective

The goal of this project is to design a complete Reinforcement Learning-based system capable of solving a particular task by processing rich visual input and interacting with the environment. The most interesting feature of the system is that it does not incorporate any prior knowledge about either the task, the environment nor the nature of actions it can possibly take. For the purpose of the project, *Breakout* was chosen as a primary problem for its simplicity and availability of an easy-to-use game emulator although the same agent could be applied to other Atari games. The algorithm improves its game playing strategy by repeatedly playing the same game thousands of times gradually getting better and scoring more points.

Achievements

In order to handle the project's complexity, it has been split into two parts by introducing a secondary problem, the Grid world, of smaller scale whose successful completion is necessary but not sufficient for solving the main task. It has helped debug, compare the performance and discuss design trade-offs associated with various *learning* routines. All the RL methods used are based on Q-learning - an off-policy, online and model-free algorithm which may be easily represented in a parametric form. It is important to note that the overall system has been designed to support modularity following object-oriented principles to allow the same agent to be used for both tasks using different input preprocessing front-ends.

Grid world

Grid world is a simple RL task used here primarily for debugging and ensuring correctness of learning routines. It serves as a testbed for more sophisticated algorithms since it may be easily solved using simpler methods. Three variants of Q-learning have been implemented: exact one for each state and action pair as well as approximate ones based on Gaussian Radial Basis Functions and neural networks.

Breakout

Atari games, used as standard RL benchmarks, produce *clean* signals and allow for performing lengthy training procedures infeasible under real world conditions. The game used in this project, Breakout, is relatively simple as it does not include a second player and uses just a small set of actions. In principle, given the *mechanics* of the game, i.e. information about ball's velocity, it is possible to create a nearly perfect player however the agent used in the experiment knows nothing about the game *structure*. Its capability of learning from visual feed makes it both really complex and powerful at the same time as it may be applied to a range of other tasks without any significant changes.

Extended testing

Despite all the precautions put in place, the debugging of a convolutional neural network has proven to be very tricky both from an algorithmic and software engineering perspective. Hence two extensions to the Grid world have been used to identify and isolate possible errors. The first one consists of two plain images - black and white - associated with two terminal rewards allowing to check whether the network is capable of learning simple patterns. The second extension tests program's RL capability through transforming the Grid world into a 2D problem by associating each state with an image from the MNIST database.

Results

The project has proven to be very challenging especially from the computational point of view requiring various code modifications aimed at either improving memory complexity or increasing neural network's training rate. Despite numerous debugging attempts, a memory leak has prevented the system from completing its lengthy training procedure by causing the system to run out of RAM. The bug has turned out to be located inside *Theano's* automatic differentiation module independent of project's source code. Nevertheless, the agent has shown signs of learning despite the completion of a fifth of intended training's length only.

Conclusions

A Reinforcement Learning system supporting common interfaces used in the research community has been built with its modular design making it easily extensible to include other learning algorithms, agent strategies, sampling methods and input types. Current implementation is capable of learning, in addition to simpler test problems, to play a 2D game although needs debugging of an open source external library. The approach taken has plenty of scope for improvement using more computational resources as well as it may be applied to many other problems.

Acknowledgements

I would like to express my huge gratitude towards my two supervisors: Prof. Zoubin Ghahramani and Dr. Matthew Hoffman for their help and support throughout an entire project.

More specifically I would especially like to thank Dr. Hoffman for his availability, day-to-day discussion and enormously helpful weekly brainstorming sessions. His insightful remarks originated plenty of invaluable discussions which often led to performance improvements.

I would also like to thank Prof. Ghahramani for his continuous high-level guidance which originated the project idea and established the overall direction of the project.

Contents

1 Introduction

1.1 Motivation

The original idea behind the project was to combine different areas of Machine Learning, namely Deep Neural Networks, Reinforcement Learning and Bayesian inference, and tackle a real world problem related to challenges outlined in Yoshua Bengio's publication [24]. The direct inspiration came from a paper [5] published by *DeepMind*, a London-based start-up focusing on Deep Learning. It describes in broad terms an algorithm capable of learning to complete several different assignments satisfactorily. Due to accumulated experience, tasks themselves do not seem to be particularly hard for humans, however the algorithm itself does not incorporate any prior knowledge about them which is what makes it particularly interesting. An agent is presented with highly-dimensional visual input only and interacts with an environment by receiving *reward* signals and reacting to them without knowing anything about the nature of actions it may take. Its training involves repeating the *assigned task* numerous times hence, due to huge complexity of real world problems, different Atari computer games were used initially as a proof-of-concept. The algorithm used by DeepMind has outperformed previous state-of-the-art approaches encouraging finding other use cases for it and investigating the algorithm further. Using visual feed as an input suggests that the same method can be potentially used in real world applications as long they may be adapted to fit the Reinforcement Learning framework.

The goal of this project is to gain deeper understanding of Reinforcement Learning and Neural Networks by developing an algorithm yielding comparable results to the published paper [5] and extending it later. The report describes the approach taken by first introducing the theory behind it and later presenting a thorough description and analysis of the practical aspects associated with implementing it.

1.2 Tasks

The project required implementing several non-trivial algorithms as well as performing a computationally-intensive training scheme. Highly-dimensional visual input and an application comprising several separate components made testing particularly difficult hence it was beneficial for debugging purposes to introduce a secondary problem of smaller scale with a known solution.

1.2.1 Primary problem

The approach described by DeepMind [5] was applied to 49 different Atari games featuring various complexity levels and goals to be accomplished. Their relative difficulty depended on several factors including whether it was played against a computer opponent or how many actions an agent could choose from. For the purpose of this report, *Breakout* was chosen for the simplicity of game objectives and a limited set of possible actions.

The goal of the game is to hit as many coloured bricks with a ball as possible. The environment allows it to bounce off bricks, walls and a paddle as shown on Figure 1. Rewards associated with each brick are the same although upper *layers* accelerate the ball slightly upon contact.[1] The agent controls the red paddle and at each frame it can decide either to stay in the same position or move left or right. A single game ends when a player misses a total of 5 balls or *scores* all the bricks. It is worth noting that the score displayed on top of the screen is irrelevant from the project's perspective as the algorithm receives reward signals straight from the environment.

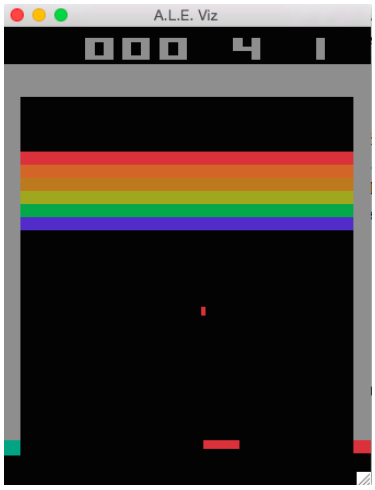

Figure 1: Screenshot from Atari's Breakout game

1.2.2 Secondary problem

Grid world is a simple problem often used to illustrate basic concepts of the Reinforcement Learning area. Its simplicity makes it suitable for testing and debugging RL algorithms as it can be solved at a relative ease.

For the purpose of this report, a 5×6 grid shown on Figure 2 was used. At the beginning of each episode, the agent is placed at a random tile and its goal is to collect a maximum reward by making a maximum of 11 moves. At each step it can move in four directions unless it is at one of the *edges*. An episode may also terminate when the agent reaches the top left corner associated with a positive reward. In fact, the optimal policy is equivalent to finding the shortest distance to the top left corner in the *Manhattan* metric.

[1] The game has no negative rewards

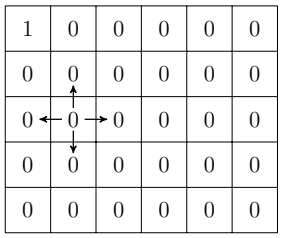

Figure 2: Grid world problem rewards layout
and scope

1.3 Project outline

A relatively long timescale for an undergraduate assignment and expected difficulty, required following a carefully devised strategy from the onset. In order to handle problem's scale and scope, the workload had to be spread evenly across the whole year gradually building up complexity by implementing new features. It was of utmost importance to split the project into several independent modules in accordance with object oriented programming principles letting individual components be modified and replaced without breaking existing functionality. Such separation also made debugging easier by allowing each component to be tested independently before integrating it with the whole system.

The outline of a high level, initial plan:

1. Build the Grid world using RL-Glue framework
2. Implement Q learning algorithm using
   (a) Exact state-action values look-up table
   (b) Linear architectures approximator with Gaussian basis functions
      i. solved using Normal equations (LSPI [8])
      ii. trained using Stochastic Gradient Descent
   (c) Neural Network approximator
3. Implement Experience Replay sampling strategy
4. Build a Convolutional Neural Network using a CPU
5. Replace Grid world's environment with signals from Atari's emulator for Breakout
6. Optimize for CUDA computing and train the model on a GPU

1.3.1 Modifications

The high level plan did not change much throughout the project although some parts took longer than expected to complete or required extra testing and bug fixing effort. An especially troublesome part was the implementation and debugging of a Convolutional Neural Network due to its high complexity and issues related to CUDA computing.

1.4 Challenges

The project proved to be challenging for several reasons, most of which had been anticipated during the planning phase. Despite the lack of prior exposure to Reinforcement and Deep Learning areas, grasping theoretical concepts was generally less difficult than solving practical issues which arose over the course of the project.

1.4.1 Problem scale

The greatest challenge of the Breakout RL task is its *scale* directly related to a highly-dimensional visual input fed into the agent which allows the algorithm to generalise well to other games without incorporating any information about their structure. In principle, game dynamics are straightforward and can be described by analysing ball's motion making it is easy to compute the optimal path of the paddle. The algorithm, however, is presented with a feed of 84×84 pixels each of which, discarding colours and limiting precision to 8 bits, can take one of 256 values leading to $256^{84*84} = 2^{8*84*84} = 2^{56448}$ distinct input states. Needless to say it is impossible to store optimal actions for each state and hence a really powerful approximator with enough parameters is required in order to provide good generalisation.

1.4.2 Software complexity

Another troublesome issue stems from an inherent complexity of combining elements from multiple theoretical areas, using libraries from sources under ongoing development and, at times, modifying someone else's source code. Despite following object oriented principles and using version control during the development, the daily-increasing complexity of the program and changing external libraries caused a few problems over the course of the project. It is worth mentioning, however, that if it had not been for design precautions taken, solving issues related to software design would have required much more effort.

1.4.3 GPU implementation

An approximator based on a Convolutional Neural Network contains a lot of parameters which need to be initialised and optimised. However, the structure of a neural network and frequent transformations of 2D data such as convolution can greatly benefit from performing them on a CUDA-enabled GPU. Graphics cards include plenty of parallel computing units especially capable of manipulating matrices. As such, running GPU-optimized code can speed up training of CNNs even 14 times [9]. Nevertheless, it was

challenging to debug and optimize the code for a GPU due to the lack of previous exposure to CUDA programming.

1.4.4 Training procedure

Optimisation of a deep neural network is a very computationally intensive task due to a very large number of model parameters. For example, the architecture used in this project contains 1638515 distinct parameters. Their robust estimate requires a significant number of samples greatly complicating model's training procedure. Despite an order of magnitude speedup through the use of a GPU, it still took around a day of computing time to obtain relatively reasonable estimates. As such, it made evaluating model's trained parameters take longer slowing down the overall development process.

2 Reinforcement Learning framework

RL is a computational approach to learning from interacting with the surrounding world. The goal of RL algorithms is to learn what to do at a particular state of the environment in order to maximise total numerical reward *collected* over the course of interaction. It must discover what actions yield the best long-term rewards by trying various sequences of them. In many problems, including Atari games, each action taken has an effect not only on immediate rewards but also on all the subsequent ones making learning more complicated. Determining which actions lead to later rewards, a so called *temporal credit assignment problem* [31], is the crux of RL.

The following subsections formulate a more rigorous description [1] of the Reinforcement Learning framework described above.

2.1 Definitions

2.1.1 The agent-environment model

An RL problem can be specified formally by introducing the *agent-environment* framework, conceptually shown on Figure 3, where the *agent* is a decision maker controlled by an RL algorithm and everything it does not have absolute control over or only interacts with is considered to be part of the *environment*. The *environment* provides the agent with its state and an immediate reward. For the purpose of this paper, the agent and environment interact with each other at discrete time steps $t = 0, 1, ...$ although in general it is possible to extend it to a continuous-time case[1].

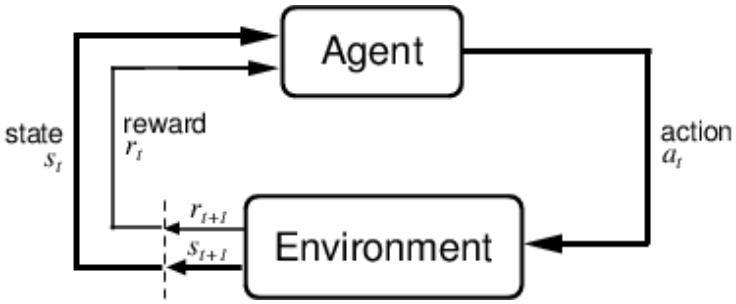

Figure 3: The agent-environment interaction, figure taken from [2]

2.1.2 State space

At each time step t, the environment provides an agent with its current *state*, $s_t \in \mathcal{S}$, where $\mathcal{S}$ is a set of possible states. For the Grid world problem, two different representations of state spaces were used:

- $\mathcal{S} = \{0, ..., 29\}$ corresponding to each of 30 tiles
- $\mathcal{S} = \{I_0, I_1, ..., I_{29}\}$, where $I_k = [a_0, ..., a_{29}]^T, a_k = 1, a_{i \neq k} = 0$ is an indicator vector corresponding to the k'th tile

In case of the Breakout task, the situation is more complicated and each state is represented by a set of last 4 consecutive game frames $\{x_t, x_{t-1}, x_{t-2}, x_{t-3}\}$ preprocessed using a function $\Phi(x)$ to reduce their dimensionality and make them suitable for CUDA-optimised convolutional operators. Hence:

- $s_t = \{\Phi(x_t), \Phi(x_{t-1}), \Phi(x_{t-2}), \Phi(x_{t-3})\}$

2.1.3 Action space

Similarly to the state space, at each time step t the agent chooses an action $a_t \in \mathcal{A}(s_t)$, where $\mathcal{A}(s_t)$ is a set of actions available at state s_t. In both tasks, $\mathcal{A}(s_t)$ is the same regardless of the state:

- Grid world - $\mathcal{A} = \{0, ..., 3\}$, corresponding to *left, right, up, down* moves respectively
- Breakout - $\mathcal{A} = \{0, 1, 2\}$, corresponding to *left, right, do not move* actions respectively

Note: environments do not have to be deterministic, i.e. taking an action a_t in state s_t may lead to several other states according to hidden transition probabilities of the *model*. Also, in the Grid world the agent cannot *step off* the Grid suggesting that $\mathcal{A}(s_t)$ is not the same for all the states. The environment implemented, however, allows all the actions to be taken resulting in the agent staying in the same position in case it tries to *step off*.

2.1.4 Reward

An essential part of the RL framework is a numerical reward signal, $r_t \in \mathcal{R}$, which the agent receives from the environment. These rewards usually cannot be attributed to any particular action but rather to a whole sequence of previous states visited and actions $(s_t, a_t), (s_{t-1}, a_{t-1}), ..., (s_0, a_0)$ taken before.

2.1.5 Returns

The objective of an RL algorithm may be more formally expressed in terms of a long-term return at time t, R_t, which is described using a standard discounting technique:

$$R_t = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{T-1} r_{t+T} = \sum_{k=0}^{T-1} \gamma^k r_{t+1+k}$$

where γ is a discount factor and the game terminates at time $t + T$. Small value of γ puts more emphasis on immediate rewards whereas $\gamma \approx 1$ makes the agent more *far-sighted*.

2.2 Policy

Policy is the probability distribution $\pi_t(s, a)$ over states and actions at time step t of selecting a particular action a while being in a state s. Formally, the goal of an agent is to learn a policy which maximises the expected discounted return over the long run.

2.3 Markov Decision Process

The Reinforcement Learning framework states that the agent makes its decisions solely based on a signal from the environment - its *state*. In particular, the state should be as informative as possible to allow the agent to choose what to do without having to consider the sequence of all previous states. In other words, it should act as a *summary* of all the past *interactions*. A state defined in such way is said to be *Markovian* [1]:

$$\Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, ..., r_1, s_0, a_0\} = \Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\}$$

meaning that its one-step dynamics allow to predict the next state and expected reward given the current state and action only. A reinforcement learning task with finite state and action spaces satisfying the *Markov property* is called a *finite Markov Decision Process*. Such assumption underlies all the Reinforcement Learning algorithms discussed and used in this report. Although it is straightforward to see that Grid world's environment is in fact *Markovian*, it is not immediately clear for Breakout. By considering each frame of the game as a separate state it is easy to see that it violates the *Markovian* assumption as it is impossible to infer ball's velocity from a single frame. In fact, it may not be possible to do so from two consecutive frames either in case the ball bounces off a paddle following the incoming trajectory. Figure 4 shows two consecutive frames where the ball is in the same position despite that it is in fact moving. Forming a single *state* from the last 4

consecutive frames, although still not entirely Markovian, offers a good approximation and can be used successfully for Atari games.

Figure 4: Illustration of Markov-violating state representation

2.4 Action-value function

The majority of Reinforcement Learning algorithms try to find an *optimal policy* by evaluating how *good* a given state is based on the notion of estimating expected long term discounted return. There are two concepts which can be used - *state* or *state-action* value functions. The latter one is generally more convenient to use as it makes selecting optimal actions easier and hence forms the basis of all the methods used in this report. In fact, using only the state value function would require a simulation to perform *policy improvement* since the agent is not given environment's transition probabilities. In other words, it does not know which state a particular action would take him to until it actually tries it.

Formally, a state-action value function is defined:

$$Q^{\pi}(s,a) = E_{\pi}\{R_t \mid s_t = s, a_t = a\} = E_{\pi}\Big\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \Big\}$$

where R_t is the expected return starting from time t and π is the followed policy. In other words, it is the expected long term return of taking action a in state s and following policy π thereafter. The task of all the RL algorithms is to find an optimal Q function, that is:

$$\forall_{a \in \mathcal{A}, s \in \mathcal{S}} \quad Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)$$

2.5 Epsilon greedy strategy

One of the more important design decisions to make when implementing an RL algorithm is the choice of a strategy for the agent to follow. It entails two aspects which need to be considered when devising one, namely *exploitation* and *exploration*. In general, the agent needs to try the actions which have been effective in the past in order to obtain more reliable

value estimates, that is to *exploit* what it already knows. At the same time it needs to take non-optimal actions as there might be a more beneficial *trajectory* through the state space about which the agent simply does not *know* yet. Epsilon-greedy is an approach which offers a trade-off between the two by taking a random action with probability ϵ and an optimal one otherwise. Usually, ϵ is initialized to a high value encouraging *exploration* initially and goes down as the game progresses placing more emphasis on *exploitation*.

2.6 Sampling techniques

The optimization routine used to train Q-function approximators is based on Stochastic Gradient Descent method. It requires, as discussed in Section 4.3.7, a *batch* of samples to perform a parameter update. The task, however, differs from other Machine Learning problems in a way that there is no *static* dataset to use for updates but new datapoints are generated in real time while playing.

2.6.1 Online

The simplest way of selecting samples for an SGD update is to use *online* updates, that is to use a batch of size 1, selecting the most recent input vector and discarding it afterwards. For smaller problems when it is easy to obtain many samples, like the Grid world, it may be an efficient way of optimising parameters although for more complicated inputs and larger models it does not work well.

2.6.2 Experience Replay

The scale of the Breakout problem renders online sampling inadequate considering how long it takes to update all parameters and generate new samples from the emulator hence a method similar to mini-batch updates is more useful. The dynamic nature of the problem, opposed to for example the MNIST challenge [26], means that the agent needs to generate and maintain its own dataset for parameter updates. Experience Replay technique tries to imitate that by keeping a history of the last N frames and drawing samples from a uniform distribution for each parameter update. Section 5.4.1 describes design trade-offs associated with implementing the ER sampler.

3 Computing the optimal Q function

There are several methods of finding the optimal Q function depending on the task involved although they all make use of the fact that a state-action value function is a solution to Bellman's optimality equation for a given policy π:

$$Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s'} P(s,a,s') \max_{a'} Q^{\pi}(s',a')$$

where $P(s,a,s')$ is the probability of a transition from s to s' by taking action a, and $Q^{\pi}(s,a)$ is stored for any combination of state and action. A similar equation can be

derived for a *state* value function, $V(s)$, which can be, in fact, expressed in a matrix form and solved by an appropriate matrix inversion [1] however it is not only infeasible for larger state spaces but also iterative methods are usually faster.

3.1 Monte Carlo methods

Bellman's optimality equation in its standard form assumes that the model of the environment, that is transition probabilities $P(s, a, s')$, is known allowing to compute the appropriate sum. For most problems the model is not known a-priori although there are ways of estimating it. Usually, however, one needs to resort to sampling-based Monte Carlo methods which assume that samples used in updates follow appropriate distributions.

3.2 Dynamic programming

As mentioned in Section 3, it is possible to calculate the solution Q^π to the Bellman optimality equation of an MDP through an iterative scheme one of which is a Dynamic Programming approach. It initializes Q^π arbitrarily and keeps updating it afterwards until a termination condition is met:

$$Q^\pi_{(k+1)} \rightarrow R + \gamma P^\pi Q^\pi_{(k)}$$

Due to P^π being a non-expansion [1], the algorithm converges to the state-action value of a given policy. The main problem of a method described above, namely *Value Iteration*, is that it only evaluates a given policy without estimating the optimal one. Hence it needs to update its policy afterwards, that is to perform *Policy Improvement*, based on newly estimated values and repeat the cycle until convergence as shown conceptually on Figure 5. Not only is prohibitively expensive for problems with larger state spaces but also it requires environment's transition model to be known which is usually not true for RL problems hence other methods are used.

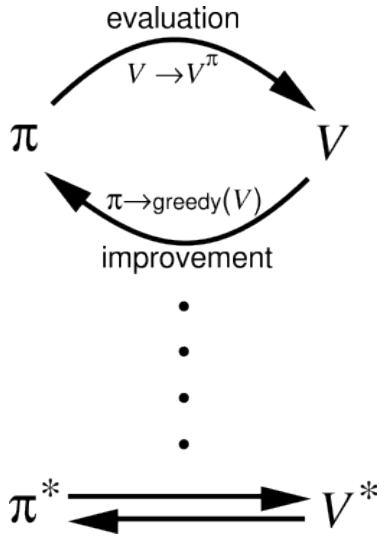

Figure 5: Generalized policy iteration, figure taken from [3]

3.3 Temporal Differences algorithm

TD algorithms combine good aspects of both DP and Monte Carlo methods forming a more suitable algorithm for the Breakout problem. They update their estimates based in part on other learned estimates without waiting for them to converge as in *Policy Iteration* method. They also learn directly from interacting with the environment without knowing neither estimating its model.

3.3.1 Q-Learning

An especially interesting method is the off-policy TD algorithm, Q-Learning, fort its ability to approximate the optimal *state-action* value function independently of the policy being followed as opposed to SARSA [1]. It ensures correct convergence as long as all the state-action values keep getting updated regularly [1]. Q-learning follows the rule:

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

In other words, at each step it tries to decrease the *temporal difference* between the current state-action estimate $Q(s_t, a_t)$ and its stochastic expected value based on the immediate reward and next state's estimated value.

4 Approximations of Q function

The Reinforcement Learning framework introduced in Section 2 described the theory behind solving an RL problem by estimating the state-action value function Q for each pair (s_t, a_t). In many practical problems, including Breakout, it is not only impossible to estimate but even to store it for every data point. A standard way of tackling such problems by expressing the value function in a parametric form, $\hat{Q}(s, a, \omega)$, usually results in reduced storage requirement by having to keep parameters ω only. Depending on the task's complexity, the choice of an approximator function and its form offers a trade-off between computational complexity and *approximating* capacity. In the limit, one may construct a function $\hat{Q}(s, a, \alpha) = \sum_{s_i,a_j} \delta(s - s_i, a - a_j)\alpha_{s_i,a_j}$ which is a sum of delta functions centred around every possible state-action pair without achieving any storage saving and being equivalent to the exact Q-learning algorithm.

This section explores the application of two approximating function families to tasks outlined in the introduction. One of them, based on linear architectures, is only used in the Grid world problem as it makes no sense applying it to the Breakout game due to the size of its state space. On the other hand, a neural network is applied to both tasks with slight architectural changes. In both cases the Q-learning update from Section 3.3.1 is replaced with minimization of the least-squares cost function with respect to model parameters [1]:

$$L(\omega) = \frac{1}{2}\sum_{i}^{n} P(s,a,s')\Big(Q^*(s'_i,a_i) - Q(s_i,a_i,\omega)\Big)^2 \approx \frac{1}{2n}\sum_{i}^{n}\Big(Q^*(s'_i,a_i) - Q(s_i,a_i,\omega)\Big)^2$$

where $Q^*(s_i,a_i)$ is the temporal differences update target which is considered constant in the parameter optimization. The approximation assumes that sample tuples were drawn from an appropriate distribution.

4.1 Linear architectures model

An approximator may be formed as a linear combination of hand-crafted basis functions $\Phi_i(s,a)$. The number of basis functions, k, is arbitrary and their forms are often adjusted to fit a particular type of a state space:

$$\hat{Q}(s,a,\omega) = \sum_{i=1}^{k}\phi_i(s,a)\omega_i = \omega^T\Phi(s,a)$$

A popular approach is to use a set of regularly distributed radial basis functions, often Gaussian, covering well the state space. Usually there is a separate set of linear coefficients for each action. Individual Gaussian basis function can be expressed as:

$$\Phi_{i,a}(s) = \exp\Big(-(s-\mu_{i,a})^T\Sigma_{i,a}^{-1}(s-\mu_{i,a})\Big)$$

$$\Phi(s,a) = \Big[\phi_{1,a_1}(s),\phi_{1,a_2}(s),...,\phi_{1,a_A}(s),\phi_{2,a_1}(s),...,\phi_{k,a_A}(s)\Big]^T$$

where A is the number of available actions and k is the number of basis functions. The parameters μ and Σ of Gaussian radial basis functions $\Phi_i(s,a)$ are usually fixed and not subject to optimization. The vector $\Phi(s,a)$ above is kA dimensional and hence the whole model has kA parameters specified by a vector ω which need to be optimized. It is possible to find optimal parameters through constructing and manipulating appropriate matrices as described in [8] however it is, in fact, equivalent to running a stochastic gradient descent algorithm on a TD-error cost function. Samples collected through interacting with the environment implicitly model their probability distribution in a similar fashion to MC methods from Section 3.1. Consider an L2 norm cost function for one sample tuple (s,a,r,s'):

$$L(\omega) = \frac{1}{2}\Big(r + \gamma\max_{a'}\hat{Q}(s',a';\omega) - Q(s,a;\omega)\Big)^2 = \frac{1}{2}\Big(r + \gamma\max_{a'}\hat{Q}(s',a';\omega) - \omega^T\Phi(s,a)\Big)^2$$

TD-estimate, $(r+\gamma \max_{a'} \hat{Q}(s', a'; \omega))$, is considered constant while optimizing with respect to ω, hence the gradient $\frac{\partial L}{\partial \omega}$ can be calculated:

$$\frac{\partial L}{\partial \omega} = \Big(r + \gamma \max_{a'} \hat{Q}(s', a'; \omega) - \omega^T \Phi(s, a)\Big)\Phi(s, a)$$

4.1.1 Grid world basis functions

Grid world's state space can be represented as a 2D map through which the agent may navigate in order to find the reward. It seems natural to describe its state-action value function with a set of 2D radial basis functions located on a regular grid in between the tiles with standard deviation large enough to provide overlap between consecutive ones as conceptually shown on Figures 6 and 7 below:

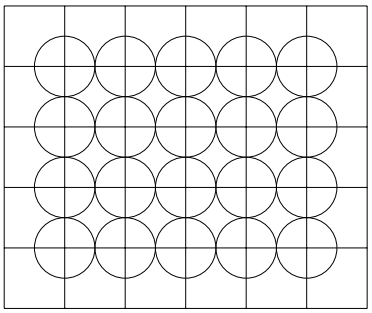

Figure 6: RBFs for the Grid world problem

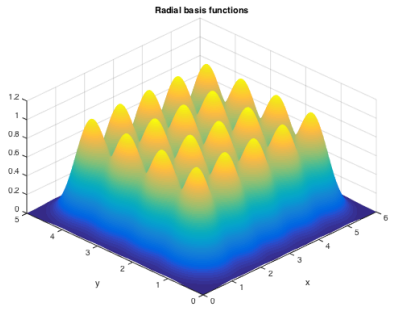

Figure 7: 3D visualisation of RBFs

4.2 Neural network for the GridWorld

Neural networks offer another approach to approximating Q-function by creating its parametric form. They are considered to be universal approximators [11] and experience a huge surge in popularity these days [29]. Hence it is interesting to include them in the comparison with other discussed methods considering the fact that their variation is also used in the Breakout task.

Note: the same cost function for a single tuple (s, a, r, t, s') is used as in Section 4.1:

$$L(\omega) = \frac{1}{2n} \sum_{i}^{n} \Big(Q^*(s'_i) - Q(s_i, a_i, \omega)\Big)^2$$

where $Q^*(s'_i, a_i, \omega)$ is the TD estimate considered to be constant in the optimization.

$$Q^*(s'_i) = \begin{cases} r_i + \gamma \max_a Q(s'_i, a, \omega) & \text{if } s'_i \text{ is a non-terminal state} \\ r_i & \text{otherwise} \end{cases}$$

4.2.1 Input preprocessing

The idea behind applying a neural network to the Q-learning algorithm is to model $Q(s, a)$ using a single feed-forward network for all actions. It is possible to take a similar approach to linear architectures and keep distinct models for each action. However, it is more appealing to use a single neural network with several outputs corresponding to each action in order to save on computation and the number of parameters. Such model takes a state s as an input and produces a state-action value for each action a at its output nodes.

It is important to notice, however, that the input to the model should not be a single decimal number corresponding to the state s as it would make it harder for the network to distinguish between distinct states. This problem is especially apparent when issues of vanishing gradients and saturating neurons are taken into account as states would differ just by a scaling factor. Instead, a better approach is to represent each state with an N dimensional binary vector:

$$f(s) = [a_1, a_2, ..., a_N]$$

There are multiple ways of mapping Grid world states into binary representations however two of them are relatively straightforward. For example, its decimal value may be expressed in a binary system thus requiring only 5 input neurons as $2^5 \geqslant 30$. Another, even simpler approach mentioned in Section 2.1.2 is to use a 30-dimensional *indicator* vector:

$$f(s = k) = [a_1 = 0, a_2 = 0, ..., a_k = 1, ..., , a_N = 0]^T$$

4.2.2 Architecture

The size and complexity of the Grid world problem does not require a large and deep network to yield a good approximation of a Q function. Hence it provides a good test bed for methods and libraries used to build neural network models.

The network used consists of three layers - input, output and a fully connected hidden one in the middle as shown on Figure 8 below.

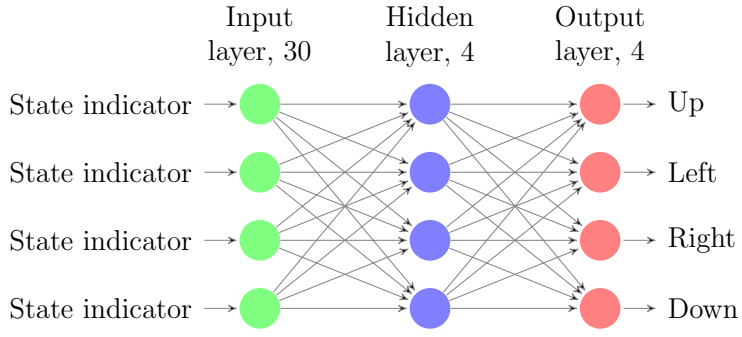

Figure 8: Neural network architecture for the Grid world

4.3 Convolutional Neural Network for Atari games

A powerful Q-function approximator based on a large neural network is particularly applicable to higher-dimensional problems with much bigger state spaces. The *visual* 2D representation of a state bears a lot of resemblance with static images which exhibit plenty of locally spatial and temporal dependencies. It has been shown in the literature that adding convolutional layers on top of a few fully connected hidden layers yields good results in classifying images [7]. In theory, such network is a special case of a fully connected one however *tying* parameters through the use of convolutional filters adds in not only translation invariance to recognizing features but also significantly reduces the number of parameters making optimization easier and require less training data.

Note: the same cost function is used as in Section 4.2.

4.3.1 Input preprocessing

The visual feed from the Atari emulator is transferred in an RGB format where each pixel is represented by three unsigned bytes corresponding to each colour. Although including all the colour channels as part of an input is possible due to the way a convolutional neural network works, it would be impractical as the colour does not usually matter in *action games*. In most of the Atari ones, especially in Breakout, the colour of pixels does not convey any *information* useful for understanding the situation on the screen. In fact, the agent would be equally knowledgeable if it was given a *black and white* representation of a game screen instead as shown on Figure 9. It is clear that the greyscale conversion preserves the *game structure* needed to play it successfully. Contours of paddles and bricks may still be recognized despite three fold reduction in input's dimensionality. The visual feed is also cropped to a square discarding useless elements such as the score counter on top and subsampled by two to reduce the dimensionality even further.

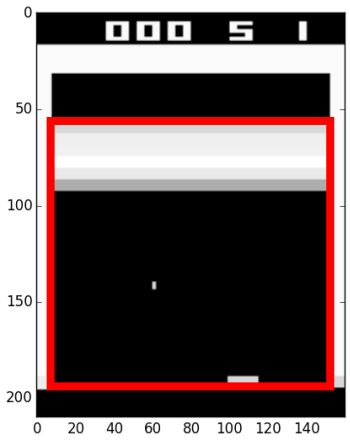

Figure 9: Greyscale display

4.3.2 Primary architecture

The initial CNN used in this project consisted of an input layer, three convolutional layers, a fully-connected hidden one and an output as shown on Figure 10. The properties of each layer used are listed in Table 1:

Layer	Activation function	Filter size	Stride	# of filters	# of nodes
First convolutional	Rectifier	8×8	4	32	N/A
Second convolutional	Rectifier	4×4	2	64	N/A
Third convolutional	Rectifier	3×3	1	64	N/A
Hidden	Rectifier	N/A	N/A	N/A	512
Output	Identity	N/A	N/A	N/A	3

Table 1: Primary convolutional neural network properties

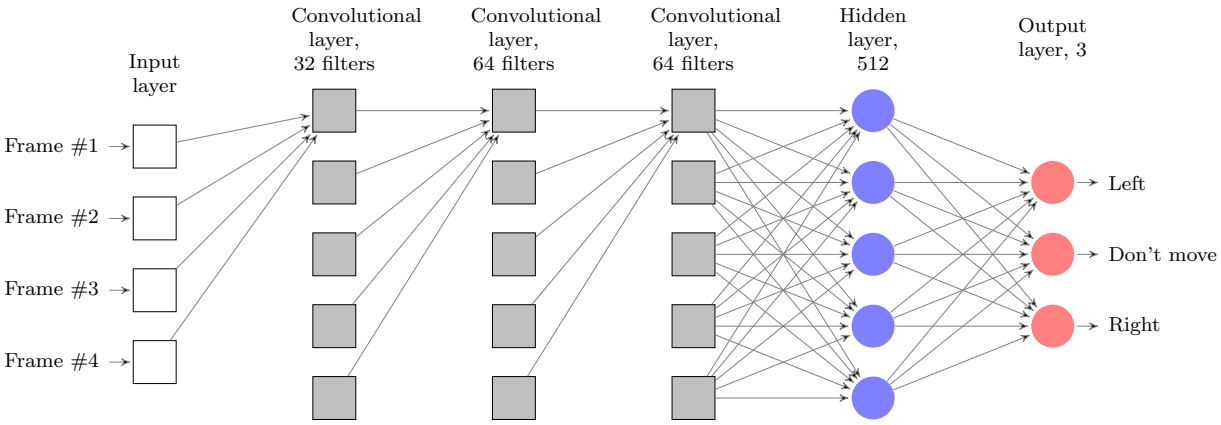

Figure 10: Breakout's primary network architecture

4.3.3 Secondary architecture

During the experiments, the initial convolutional neural network turned out to be quite complex and computational requirements of its training procedure exceeded available resources hence a second architecture was also considered. Compared to the primary one, it only had two convolutional layers and fewer nodes in the fully-connected hidden one as shown on Figure 11. The properties of each layer used are listed in Table 2:

Layer	Activation function	Filter size	Stride	# of filters	# of nodes
First convolutional	Rectifier	8×8	4	16	N/A
Second convolutional	Rectifier	4×4	2	32	N/A
Hidden	Rectifier	N/A	N/A	N/A	150
Output	Identity	N/A	N/A	N/A	3

Table 2: Secondary convolutional neural network properties

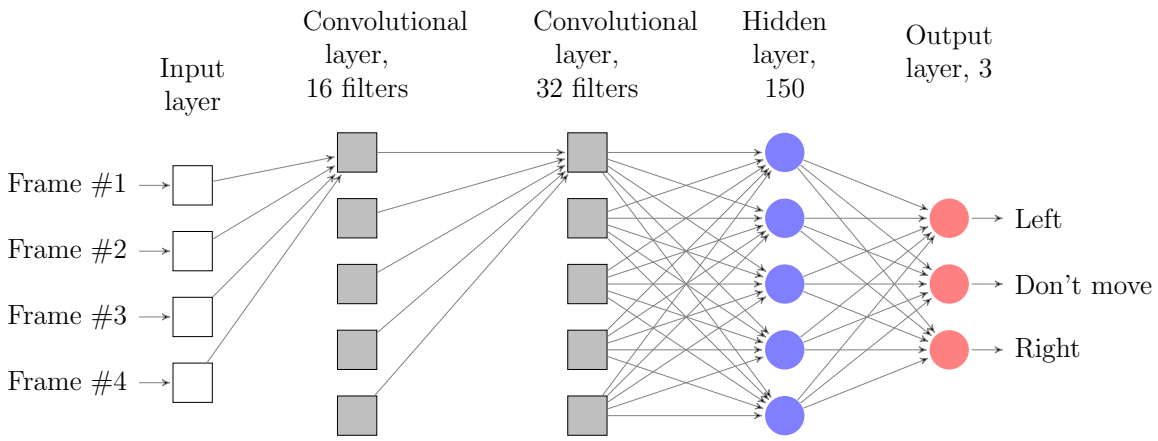

Figure 11: Breakout's secondary network architecture

4.3.4 Convolutional layers

Convolutional layers at the bottom of a neural network are essential in achieving their high performance in recognizing or classifying pictures. Each convolutional layer consists of a set of 2D *filters* which are convolved with an input image to find *areas* most similar to the filter itself. The response at a particular pixel (i, j) to the k'th filter is given:

$$z^k_{i,j} = (W^k * x)_{i,j} + b_k$$

where W^k are filter weights, b_k is filter's bias and $*$ is a convolutional operator.

4.3.5 Rectifier non-linearity

A standard neural network's node, in addition to calculating a weighted sum of its inputs and adding a bias, applies a, usually non-linear, *activation function* in order to improve its performance or speed up the training process. There are many widely used activation functions depending on an application however the literature suggests that a rectifier non-linearity yields the best performance for convolutional neural networks and 2D input data [7]:

$$f(a) = \max(0, a)$$

4.3.6 Parameter initialization

Multi layer perceptrons were first introduced in 1980s and quickly became popular when people discovered performance and generalisation gains arising from adding a hidden layer. The discovery led many people to believe that adding more layers would yield even greater improvements. These more complex models, however, did not deliver satisfactory performance due to problems associated with their training and initialization [18][23] resulting in parameters converging to bad local minima. Only recently deep learning models became popular due to a breakthrough in their initialization [13] giving rise to many successful applications. For the purpose of this report, two methods of random parameter initialization are considered:

- $\omega \sim \mathcal{N}(0, \sigma^2)$ where sigma depends on characteristics of each layer [23]
- $\omega \sim \mathcal{U}[-a, a]$ where a is a preset constant

4.3.7 Stochastic gradient descent with a mini-batch

Neural networks, although powerful, are particularly hard to train and prone to many problems including getting stuck in suboptimal local minima of a cost function [18]. They usually require expert knowledge in choosing the architecture as finding optimal *hyperparameters* is prohibitively expensive in a computational sense. On the other hand, a very large number of model parameters and huge cost of evaluating them renders more sophisticated global optimisation methods, for example, evolutionary algorithms simply infeasible. Gradient descent with mini-batches is a standard method used for training neural networks however, in case of this report, its stochastic version is more suitable. The standard update form at the $(k+1)$'th iteration is the following:

$$\omega^{(k+1)} = \omega^{(k)} - \eta \sum_{n=1}^{N} \frac{1}{N} \frac{\partial L(x_n)}{\partial \omega}\bigg|_{(k), x_n \sim \mathcal{X}}$$

where η is a learning rate, x_n are input samples drawn from distribution $\mathcal{X}$ and L is the cost function to be optimized.

4.3.8 RMSProp

There are many suggested improvements to a basic stochastic gradient descent algorithm aimed at providing faster learning. Some of them make use of previous updates through a momentum term, utilize second order information or adjust learning rates for each parameter individually. RMSProp [6] is one of such techniques which works reasonably well for neural networks. It adjusts a particular learning rate using a running average g_1 of its squared gradients as shown below:

$$\eta^{(k+1)} = \frac{\alpha}{\sqrt{\gamma_0 + g_1^{(k+1)}}}$$

$$g_1^{(k+1)} = \rho_1 g_1^{(k)} + (1 - \rho_1)\Big(\frac{\partial L}{\partial \omega}\Big)^2$$

It is worth noting that there is an inverse relationship between gradient's magnitude and its learning rate. γ_0 coefficient determines a maximum learning rate and ensures that the square root is proper. The running average is initialized as $g_1^{(0)} = 0$.

Note: mini-batch notation has been omitted here for clarity. Gradient in the expression for the running average is evaluated using parameters from k'th iteration.

4.3.9 Extended RMSProp

It may be useful for certain problems to include an element which would have a similar effect to introducing a *momentum* term. It can be done by using a running average of gradients g_2 initialized in a similar fashion to the regular RMSProp ($g_2^{(0)} = 0 \wedge g_1^{(0)} = 0$):

$$\eta^{(k+1)} = \frac{\alpha}{\sqrt{\gamma_0 + g_1^{(k+1)} - \left(g_2^{(k+1)}\right)^2}}$$

$$g_2^{(k+1)} = \rho_2 g_2^{(k)} + (1-\rho_2)\Big(\frac{\partial L}{\partial \omega}\Big)$$

The expression in the denominator may appear at first to become negative for certain values of $\{g_1, g_2\}$ however it can be proven that it is strictly positive for $\rho_1 = \rho_2$. Letting the i'th previous gradient be β_i allows the running averages to be expressed:

$$\begin{aligned} g_2^{(k+1)} &= \rho_2 g_2^{(k)} + (1-\rho_2)\beta_{(k)} = \rho_2(\rho_1 g_2^{(k-1)} + (1-\rho_2)\beta_{(k-1)}) + (1-\rho_2)\beta_{(k)} = ... \\ &= (1-\rho_2)\beta_{(k)} + \rho_2(1-\rho_2)\beta_{(k-1)} + \rho_2^2(1-\rho_1)^2\beta_{(k-2)} + ... \\ &= (1-\rho_2)\beta_{(k)} + \sum_{i=1} \rho_2^i (1-\rho_2)^i \beta_{(k-i)} \end{aligned}$$

$$\begin{aligned} g_1^{(k+1)} &= \rho_1 g_1^{(k)} + (1-\rho_1)\beta_{(k)}^2 = \rho_1(g_1^{(k-1)} + (1-\rho_1)\beta_{(k-1)}^2) + (1-\rho_1)\beta_{(k)}^2 = ... \\ &= (1-\rho_1)\beta_{(k)}^2 + \rho_1(1-\rho_1)\beta_{(k-1)}^2 + \rho_1^2(1-\rho_1)^2\beta_{(k-2)}^2 + ... \\ &= (1-\rho_1)\beta_{(k)}^2 + \sum_{i=1} \rho_1^i (1-\rho_1)^i \beta_{(k-i)}^2 \end{aligned}$$

The learning rate denominator becomes:

$$\sqrt{\gamma_0 + (1-\rho_1)\beta_{(k)}^2 + \sum_{i=1} \rho_1^i (1-\rho_1)^i \beta_{(k-i)}^2 - \Big((1-\rho_2)\beta_{(k)} + \sum_{i=1} \rho_2^i (1-\rho_2)^i \beta_{(k-i)}\Big)^2}$$

Hence the following inequality needs to be satisfied for the square root to be proper:

$$\Big((1-\rho_2)\beta_{(k)} + \sum_{i=1} \rho_2^i (1-\rho_2)^i \beta_{(k-i)}\Big)^2 < \gamma_0 + (1-\rho_1)\beta_{(k)}^2 + \sum_{i=1} \rho_1^i (1-\rho_1)^i \beta_{(k-i)}^2$$

Placing a constraint $\rho_1 = \rho_2$ allows it to be reformulated:

$$\Big(\sum_{i=0} \kappa_i \beta_{(k-i)}\Big)^2 < \gamma_0 + \sum_{i=0} \kappa_i \beta_{(k-i)}^2$$

and knowing that $\gamma_0 > 0$ and $\sum_{i=0} \kappa_i = 0.05 + \frac{0.0475}{1-0.0475} \approx 0.00013 < 0$ leads to:

$$\Big(\sum_{i=0} \kappa_i \beta_{(i)}\Big)^2 \leqslant \sum_{i=0} \kappa_i \beta_{(i)}^2$$

$$\Big(\sum_{i=0} \kappa_i \beta_{(i)}\Big)^2 \leqslant \Big(\sum_{i=0} \kappa_i \beta_{(i)}^2\Big)\Big(\sum_{i=0} \kappa_i\Big) \leqslant \sum_{i=0} \kappa_i \beta_{(i)}^2$$

arriving at a familiar Jensen's inequality for non-negative weights κ_i and convex function $f(x) = x^2$ which proves that learning rate is always proper:

$$\Big(\frac{\sum_{i=0} \kappa_i \beta_{(i)}}{\sum_{i=0} \kappa_i}\Big)^2 \leqslant \frac{\sum_{i=0} \kappa_i \beta_{(i)}^2}{\sum_{i=0} \kappa_i}$$

5 Experimental methods and implementation details

The computational nature of a project required a slightly different definition of an *experiment* which included training and evaluation procedures. Each model and its set of *hyperparameters* were scrutinised on their convergence rate and final performance in comparison to a precomputed reference model.

5.1 ALE Atari emulator

The feasibility of the project was dependent on the ability of an agent to interact with appropriate environments and, through doing so, learn how to do it in a return-maximising way. Atari emulator, ALE [14], provides a convenient yet efficient way of simulating the environment. It allows the agent to play any Atari-compatible game as long as its *bin* file is supplied. Hence the agent may be easily tested on a variety of games exhibiting different characteristics from simple ones like *Pong* up to more complicated ones like *Raid*. Its C++ source code is available online on GitHub [15] allowing developers to easily modify it according to particular needs. One of its biggest advantages comes from the fact that it implements the *RL-Glue* interface allowing to separate the environment from the learning algorithm. It also enables agents to be written in other languages provided that an appropriate interface porting exists.

5.2 RL-Glue

RL-Glue is a standard interface adopted by the AI and RL research communities which implements the agent-environment model discussed in Section 2.1.1. It provides skeleton implementations in different languages which are guaranteed to be cross-language compatible as, for example, agent written in Python may interact with ALE environment developed in C++. Following the theoretical model, it requires implementations of the agent, experiment and environment.

5.2.1 Agent

The *agent.py* file is responsible for interacting with the environment and learning from its experience. At each time step, it receives an *observation* and immediate reward from the environment. Depending on a task, an observation is either a single number $s \in \{0, ..., 29\}$ for the Grid world or a dump from emulator's RAM memory which contains an array of pixels in an RGB format as well as some other information to be discarded. In response, it chooses an action and sends it over to the environment.

5.2.2 Experiment

The *experiment.py* file is responsible for setting up a framework for training and evaluation. It specifies how many training episodes are run within each epoch as well as how often agent's performance is evaluated. In case of the Grid world, a single episode consisted of at most 11 steps whereas in Breakout each episode was played till the agent lost while a training epoch consisted of a total of 50000 frames played.

5.2.3 Environment

The *environment.py* component is responsible for simulating everything the agent is *surrounded* by and may interact with. It creates a model which keeps track of and modifies its state based on *actions* taken by the agent also providing it with immediate rewards. It may be stochastic however both the Grid world and ALE are deterministic in their nature.

5.3 Agent program design

The main goal of the project is to create an algorithm capable of learning how to traverse through the Grid world or play Breakout efficiently in order to collect highest rewards. The responsibility for coming up with the optimal policy to follow lies solely on the agent hence making it the most important and complex part of the project. In order to break-down its complexity into manageable chunks and support two tasks simultaneously, it is crucial for the program to be designed in a *modular*, object-oriented way. The agent may be split into four main, independent components as shown on Figure 12. Designing each component as a separate abstract class with inherited various implementations allowed for easy switching between them without making significant changes to the *agent.py* program:

- Q-learning algorithm - *Q Approximator*
- Agent's exploration and exploitation strategy - *Epsilon scheduler*
- Interaction data history - *Sampler*
- Input preprocessing front-end - *Observation preprocessor*

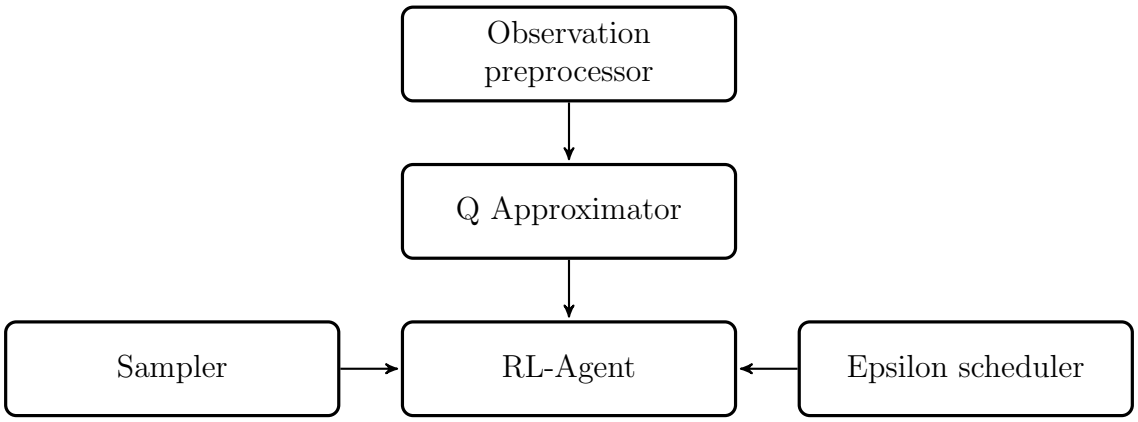

Figure 12: High level diagram of a program's structure

5.3.1 Q Approximator

Task	Approximator
Grid world	Dynamic Programming
	Linear architectures RBFs
	Neural network
	Convolutional neural network
Breakout	Convolutional neural network

Table 3: Q function approximators implemented

5.3.2 Sampler

The main task of the *Sampler* class family is to create and maintain a dataset of past sample tuples (s, a, r, t, s') collected at each time step for later use in Q-learning. It may be queried for a set of samples at any time by the agent in order to perform stochastic parameter updates. It is important to note that a Q-learning algorithm makes use of a TD-estimate $(r + \gamma \max_{a'} Q(s', a'))$ in gradient descent however the *Sampler* should not store its value as part of a tuple. The $\max_{a'} Q(s', a')$ estimate needs to be evaluated at the time of the parameter update but kept constant while optimizing.

For the purpose of this report, four different sampler types were used depending on the application as listed in Table 4:

Type	Storage	Retrieval
Finite	Most recent N	Most recent $k \leqslant N$
Infinite	All interactions	Most recent or random k
Online	Most recent one	Most recent one
Experience replay	Most recent N	Random k

Table 4: Sampler types implemented

5.3.3 Epsilon scheduler

The importance of agent's strategy in learning is discussed in Section 2.5 and the *epsilonSchedule* class family implements the ϵ-greedy strategy. Its responsibility is to decide how the epsilon parameter should vary over time as the agent gains *experience* of the *environment*. For the purpose of this report, two schedules were implemented although they can easily be extended to include other ones such as quadratic or even use a strategy different from ϵ-greedy.

Type	Formula
Constant	$\epsilon(t) = \epsilon_0$
Linear	$\epsilon(t) = \min\left(1, \max\left(\epsilon_0, 1 - \frac{(t-t_0)(1-\epsilon_0)}{t_1-t_0}\right)\right)$

Table 5: Epsilon schedules implemented

Note: Linear schedule starts off at 1 till $t = t_0$ and is annealed to reach ϵ_0 at $t = t_1$ and stay constant thereafter.

5.3.4 Observation preprocessor

The *observationPreprocessor* class family is a front-end to the designed system and implements input preprocessings described in Sections 4.2.1 and 4.3.1. As shown on Figure 12, it actually belongs to the *Q Approximator* class instead of the *RL-Agent* since it is specific to the approximator model used. Such separation of components proved beneficial when halfway through the project ALE emulator's author commited a modification switching visual feed's colour format from NTSC to RGB which required altering just a few lines of code to accomodate the change.

Task	State description
Grid world	tile number indicator vector
Breakout	concatenated frames

Table 6: Input preprocessors implemented

5.3.5 Version control

One of the challenges associated with the chosen project was its inherent complexity due to the number of various components needed to build an entire system and different software packages to be used. Hence managing complexity over project's time span was necessary for its successful completion. A particularly useful tool in running software projects which are worked on by multiple teams at once is version control. Although the program was developed by one person, implementing different components simultaneously and transferring code onto a computing unit was made much easier through using Git and hosting the code base on a private GitHub account. It also sped up the debugging process by tracing a history of commits available both online and locally on individual machines.

5.3.6 Test driven design

Another useful concept while building a multi-component software system is a test driven development approach which allows to isolate bugs and tackle system integration problems more easily. Although none of the Python's automated testings were used as part of the project, more informal tests were used extensively to check individual components both from purely software-correctness perspective as well as from Reinforcement Learning algorithmic point of view. In fact, the introduction of a secondary task - the Grid world - is an exemplary of a test driven approach. In addition, any changes made to existing functionality or an internal implementation of a given component were tested against its previous version.

The modular design of an entire system made it easier to debug the most complex algorithmically part of the project that is of convolutional neural network based Q learning. For instance, observation preprocessing component allowed to *suppress* the visual feed from ALE and replace it with a set of predefined simple patterns with associated rewards. In order to check if the network was capable of recognizing them each sample was flagged as terminal to remove the effect of the Temporal Differences estimates. Figure 13 shows a Q value plot for two of the predefined patterns with two possible actions each. Two input pairs were associated with a reward of 55 and the remaining ones with 1. The network converges for both of them with relative ease exhibiting stochastic fluctuations around target values due to the ϵ-greedy strategy.

Note: the legend hides an initial part of a graph however the more important fact is its convergence later. Also, the model did not enforce positive outputs hence Q value became negative a few times despite the lack of negative rewards.

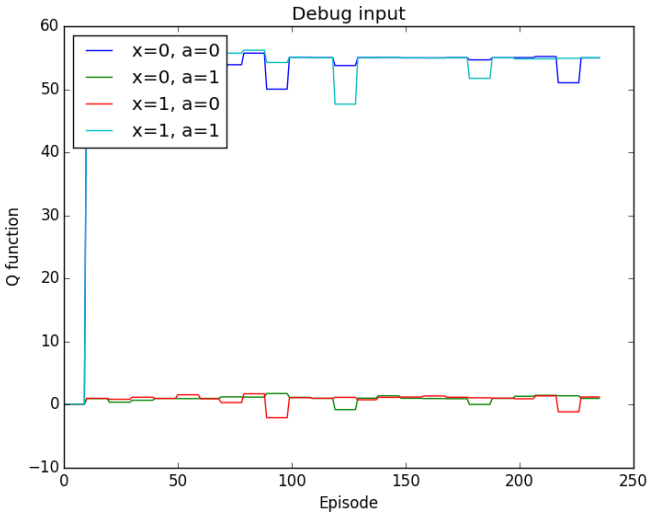

Figure 13: Q function estimates of predefined patterns

5.4 Performance improvement modifications

5.4.1 Sampler

There are a few possible ways of storing interaction data internally trading off storage space and time complexity of retrieval. Throughout the course of the project the internal implementation changed several times to meet gradually more demanding performance requirements. Initially, the *Sampler* stored (s, a, r, t, s') tuples where s, s' represent preprocessed states corresponding to concatenated four display values which had no *overlapping* observations. In the meantime, it turned out that the computing unit ran out of memory much sooner than needed. A quick inspection of the initial method suggested tuples (s, a, r, t) to be stored instead effectively doubling storage capacity as (a, r, t) elements take up negligible amount of space compared to the visual feed. Despite being able to store 350000 distinct tuples, the algorithm still yielded unsatisfactory results as shown below. Figures 14 and 15 indicate that the sampler reached its full capacity at the same time as the algorithm *finished the exploration* phase with epsilon reaching 0.1. Q-values of a heldout test set reached its peak about 1000 episodes later only to drop down to around 3 after 4000 episodes corresponding to a naive policy of waiting in one corner and resulting in scoring 3 bricks. It is important to note that by that time all the *exploration* samples were replaced by those obtained with epsilon set to 0.1 suggesting that the algorithm needed to use a longer history of more diverse samples. Hence the final modification of *Sampler's* internals changed the basic storage unit to a tuple of (d, a, r, t), where d is a single frame displayed allowing for states $s = [d_1, d_2, d_3, d_4]$ to *overlap* by sharing frames, for example $s_1 = [d_1, d_2, d_3, d_4]$ and $s_2 = [d_2, d_3, d_4, d_5]$. Such change effectively further quadrupled the storage capacity since each *observation* d could be part of four distinct states allowing a history of most recent 10^6 frames to be stored on a computing unit used. However it is important to be careful while selecting random samples from such dataset since a *state* should be composed of samples from the same episode or, in other words, should not include a terminal frame unless it is located at the end.

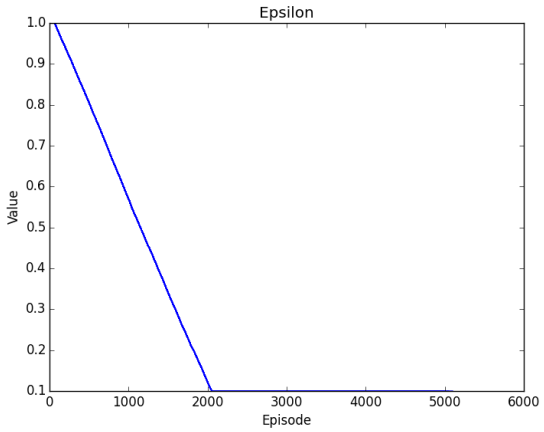

Figure 14: Epsilon schedule

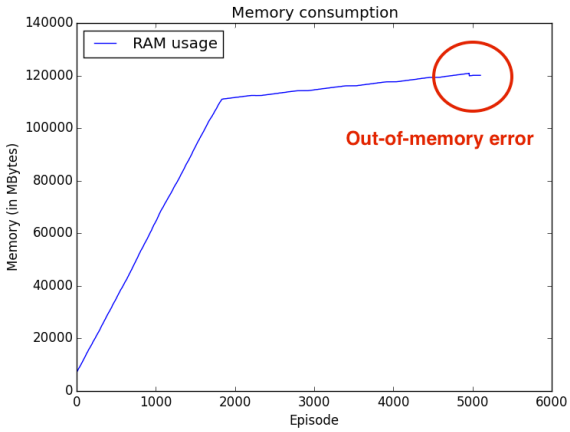

Figure 15: Memory consumption

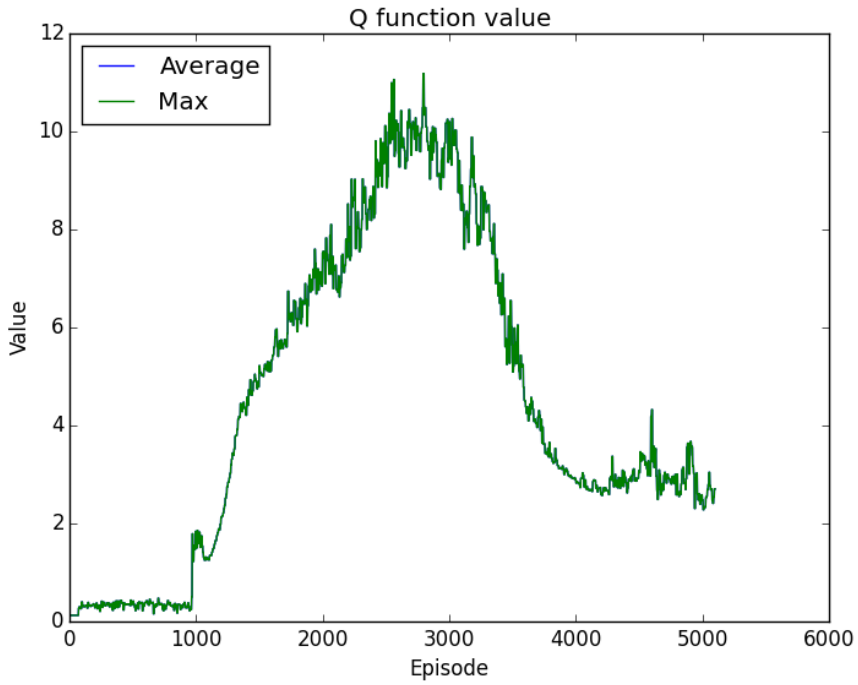

Figure 16: Q value estimate with insufficient training data

5.4.2 Observation preprocessor

During the design stage two image processing libraries were considered for the interpolation routine - *Scipy* and *OpenCV*. Both of them produced similar images as shown on Figures 17 and 18 however the algorithm proved to be very sensitive to input's preprocessing and yielded better results when the former method was used. This is probably due to the fact that the latter interpolator *smeared* the edges more making it harder for the algorithm to discern contours.

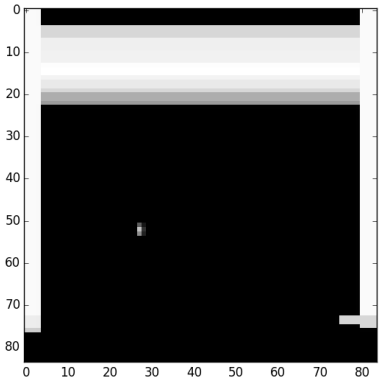

Figure 17: OpenCV interpolation

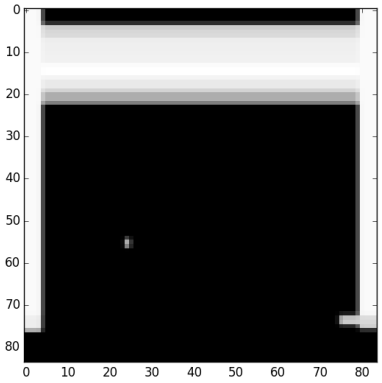

Figure 18: Scipy interpolation

5.4.3 Further convolutional neural network testing

The final stage of the project - training of the whole system - turned out to be more challenging than expected due to a computational expense and scarcity of useful diagnostic tools. It was particularly hard to spot any anomalies in the *exploration* phase at the beginning of training hence requiring several hours of computation before making any changes to the source code. Lacking satisfactory results, another diagnostic measure was employed in order to disambiguate errors caused by the neural network training and the Q learning algorithm. A natural approach was to extend the Grid world problem so that it preserved its *dynamics* and could be represented by visual input. An easy way of doing so was to associate each state with an image of, for example, MNIST digits [26] as shown on Figure 19. For simplicity, a smaller 2×2 grid was used with states represented by digits from 0 to 3 whose optimal state-action values were easily obtained.

Introducing the extra diagnostic measure allowed to uncover a few smaller bugs in the code increasing confidence in the program being capable of solving the primary task. Figure 20 shows that the neural network used *learns* to solve such simple task very quickly and with sufficient precision. Fluctuations of the error value at convergence are expected due to stochasticity introduced by linearly annealed epsilon coefficient to 0.1.

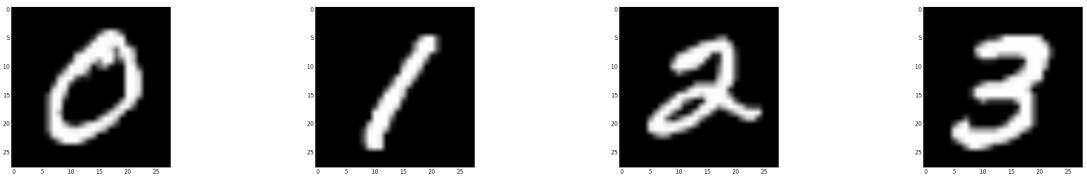

Figure 19: MNIST visual representation of Grid world states

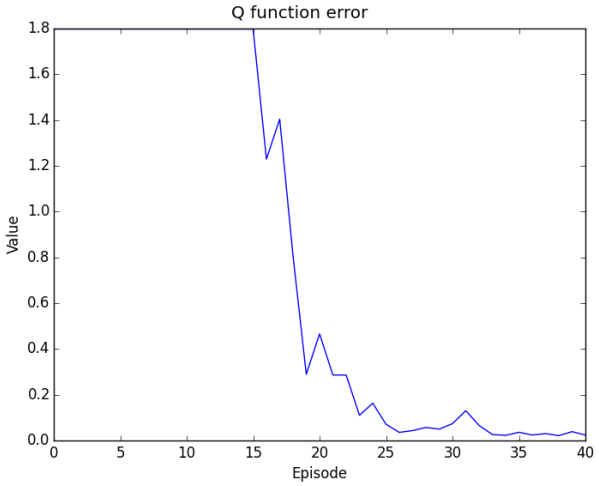

Figure 20: Q function error of MNIST Grid world

5.4.4 Memory consumption

The main challenge from the software perspective arose directly from Breakout's high-dimensional input and hence a big neural network needed. In order to obtain robust coefficient estimates, a large number of training samples and parameter updates was needed. Considering that each frame in its raw form took around 100 kB, a target history length would require around 96 GB of memory. Preprocessing each frame brought it down to around 7.5 GB of RAM memory allocated purely for storing samples excluding any overhead resulting from using *Numpy* objects. Huge memory requirements required a lot of careful planning while implementing sample-processing classes to optimize for wasteful allocation of Numpy objects while passing them from function to function. In order to speed up parameter updates, every few thousand frames a large set of randomly selected samples - around 3.6 GB - was transferred onto a GPU minimizing the number of costly transfers from RAM memory. The update frequency and size of a single dataset moved to the GPU offered a trade-off between parallel computation advantages and time overhead needed for the transfer. It turned out that using less frequent but larger updates made better use of GPU's time and resulted in faster computation overall. The problem was, however, that each out-of-order *slicing* of a Numpy array created a new one generating a short-lived spike in memory consumption which could potentially cause a crash if the system was running near its capacity. In addition, other parts of the system - RL-Glue components and interface, Theano objects and the symbolic graph as well as regular OS processes also required access to RAM memory. Overall, the whole system was projected to stay in a dynamic balance using up almost all 15 GBs available.

It is also important to note that using CUDA to speed up computation required several other modifications to the code including ensuring that Numpy arrays were C-contiguous before transferring onto the GPU.

5.4.5 Memory leaks

Designing a multi-component software project handling and processing large amounts of data required careful planning aimed at stability. It was important to make sure that the program did not crash because of an avoidable bug having completed, for example, 20 out of 22 hours of computation. There are a few issues which needed to be taken into account in order to ensure uninterrupted service however, given that Amazon AWS Cloud computing was used, the computational unit was assumed to be stable. Hence all the emphasis was placed at making the application stable. Processing a lot of data put extra pressure on the memory system as described in Section 5.4.4 which mentioned a huge memory footprint of *necessary* data storage components providing a lower bound on the usage. However, for long-running processes it was essential to carefully scrutinize the program and eliminate any potential memory leaks which could easily accumulate and destabilize the system. In fact, such situation may be observed on Figure 22 where the usage grows quickly while new samples are being added to the Sampler before reaching its capacity after 250 episodes. In theory, the plot should have leveled off afterwards as the most memory-hungry component did no longer expand. Nevertheless, the footprint kept growing steadily at a substantial rate of a quarter of the previous one despite much lower memory pressure eventually causing the system to run out of memory. It turned out that the extra memory pressure was caused by neural network parameter updates leaking around 10 MB per a cycle of 10^4 updates.

As a matter of fact the system started to learn better policies after implementing modifications which allowed to make use of a larger number of history samples described in Section 5.4.1. For example, Figure 21 shows rewards achieved on *performance runs* at the end of each epoch where the agent reached a reward of 17 after around 3 million frames played. Visual inspection of the agent's playing strategy and a quick look at state-action values calculated in real time indicated that the algorithm improved on its estimates substantially compared to the initial random state. Unfortunately the memory leak caused the system to run out of resources and crash after 3.5 million frames suggesting that fixing it may lead to better performance.

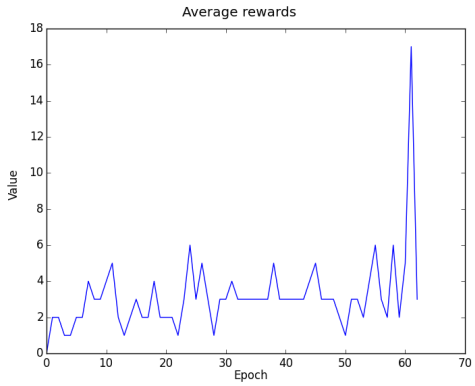

Figure 21: Reward graph

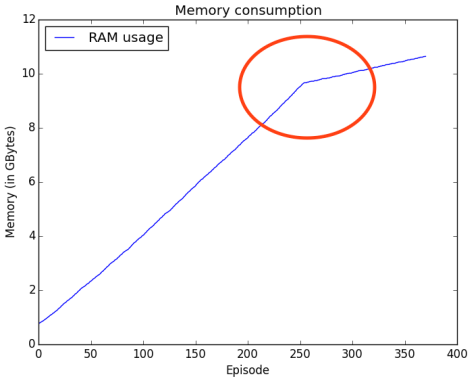

Figure 22: Memory leak

5.4.6 Further memory usage optimization

Thorough inspection of project's source code revealed a few unnecessary allocations of multidimensional *Numpy* arrays used for storing interaction samples and generating random mini-batches used in stochastic gradient descent. In fact, garbage collection of these was taken care of by the system hence turning out not to be a major issue. Further analysis of other components showed that *matplotlib* library used for graph plotting leaked around 1MB per each figure saved to a file. As a solution, the data needed for plotting was saved to a disk after each episode and plotted in a separate process afterwards. Figure 23 shows that the memory consumption plot became more regular with each step-like increase corresponding to network's parameter updates hence decreasing the overall memory leakage. Further attempts to isolate the memory leak localised the bug to be either in data transfer from RAM onto GPU's memory or in *Theano*-based parameter updates. As it turned out, disabling *backpropagation* of weight updates resulting in the application continuously transfer new samples onto the GPU led to a *flat* long-term memory footprint as shown on Figure 24 identifying the leakage source to be in parameter updates.

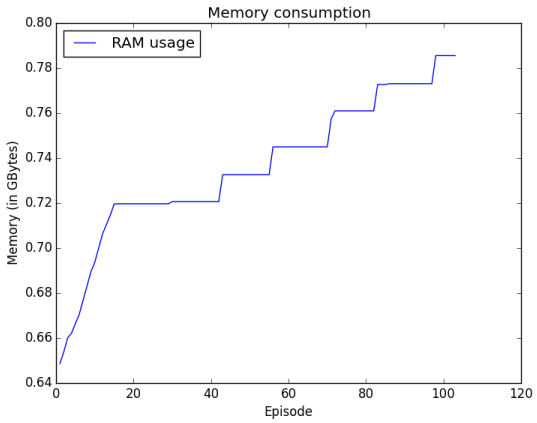

Figure 23: No graph plotting

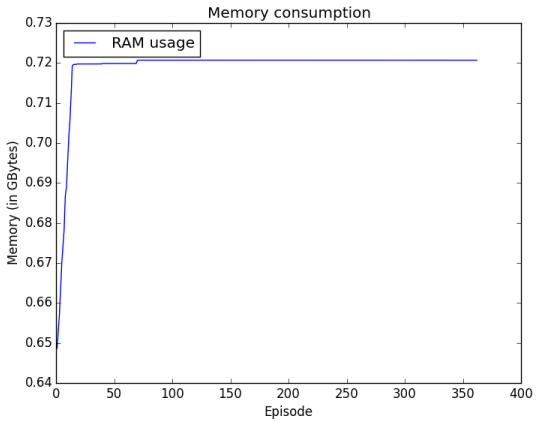

Figure 24: No parameter updates

5.4.7 Convolutional neural network libraries

Identification of a memory leak caused by *Theano's automatic differentiation*-based parameter updates allowed the debugging effort to focus on a very small part of the entire program. Nevertheless, large dependence on a continuously-developed external library made fixing the problem particularly difficult without knowing its internals. As such, one possible solution attempt was to use a different library for convolutional operations out of multiple available online [19]. As it turned out, two libraries used - *cuDNN* and *convnet* - resulted in growing memory consumption as shown on Figures 25 and 26. Both implementations improved system's performance compared to the standard Theano implementation however *convnet* reduced the memory leak in half and hence was used in all later experiments improving the situation slightly.

Note: sampler's capacity was decreased to emphasise the memory leak effect. Also, vertical scales of Figures below differ indicating memory leak's reduction.

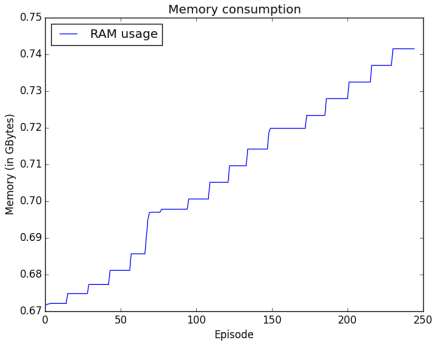

Figure 25: *convnet* memory consumption

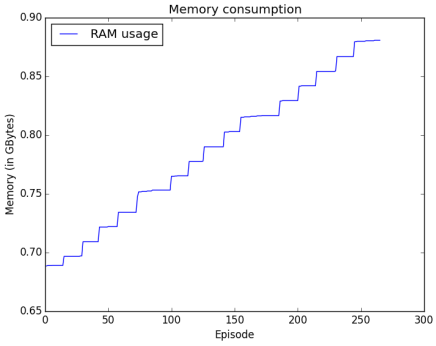

Figure 26: *cuDNN* memory consumption

5.4.8 Issues with Theano's GPU backends

Further investigation of the memory leak and consultation with developer forums [27] suggested that the bug might have been hidden inside Theano's symbolic graph or GPU-shared memory handling independent of project's source code. Following the advice found on the forum to switch to the newer GPU backend under development, *libgpuarray* [28], resulted in the application being completely unstable. Running example MNIST code found online also output multiple GPU memory errors confirming that the newer backend was not ready to use yet.

5.5 Main languages and libraries used

5.5.1 Python

The majority of development was done in Python due to its ease of use as well as the availability of widely used numerical, scientific and data processing packages kept up to date by RL and data science communities. It also made running the same code on a GPU

easier thanks to existing ports compatible with CUDA drivers and active communities sharing development problems and debugging hints.

5.5.2 Theano

In principle it is relatively straightforward to implement a convolutional neural network to perform forward computation and define correct backpropagation gradient updates. However, not only does the size of a problem render such approach infeasible due to computational inefficiencies but also it is not the main goal of the project. Instead, using the *Theano* library for Python is a much better approach from both performance and ease of development perspectives. It maintains a symbolic expressions graph providing automatic differentiation and performing behind-the-scenes optimizations including dynamically generated, faster C code. It also supports a *transparent memory* model taking care of transferring data onto a GPU.

5.5.3 C++

The only part of the project which used C++ was a modification of ALE to generate a Q function test set. Nevertheless, C++ should be considered for future extensions if Python's interpretable nature proved to be a performance bottleneck.

5.6 High performance computing

The anticipated, computationally intensive nature of the project proved to be challenging and made a few changes to the original project plan necessary. The majority of development work, Grid world experiments and simple bug fixing of Breakout's model was possible to do on a Macbook Air. However, training a convolutional neural network of such scale required a more powerful system especially capable of handling images and enabling parallel computation thus suggesting the use of a GPU instead.

5.6.1 CUDA

CUDA is the most common and widely used API for general processing on high-end GPUs developed by Nvidia. It has been actively developed for several years now allowing the technology to become mature and user-friendly enough to gain traction. Initially, one of the computers in the lab was supposed to be used for experiments however it turned out that its hardware was too old to find drivers compatible with other libraries used and an alternative solution had to be found.

5.6.2 Amazon AWS Cloud

Amazon's Elastic Compute 2 cloud service gives easy access to high performance, on-demand computing making it very easy to scale up or down by adding new processing instances depending on the workload. Most instance types include high-performance CPUs and only recently a new instance type was added - *g2.2xlarge* [16] - which features an up-to-date GPU with 15 GBs of RAM and a fast SSD drive.

5.6.3 Debugging GPU code

Moving computation and model training onto a GPU turned out to be quite tricky due to the lack of previous exposure to programming on CUDA. It was particularly important to make sure that parallel computation on a GPU had access to all the data it needed in its local memory without having to query the CPU as it would have slowed down calculations significantly. A considerable effort was put into debugging the initially underperforming code due to scarcity of user-friendly GPU debugging tools. In the end it turned out that one of the convolutional operators in *Theano* had a bug preventing it from performing efficient computation using CUDA and hence required using a diffferent, Nvidia provided, *cuDNN* library.

5.7 Diagnostic routines

Despite using high performance computing units featuring powerful CUDA-capable GPUs, the computational challenges were still significant considering that an agent needed to play around 10 million frames corresponding to roughly 150 hours of uninterrupted playing time. Factoring in time needed to perform stochastic parameter updates it turned out that the algorithm needed around 15-20 hours of computing time to yield relatively reliable Q function value estimates showing signs of *learning*. A long training procedure increased the development iteration time considerably putting more emphasis on careful code improvements before starting a new training procedure. In fact, it was really hard to tell if the algorithm had a bug or if it was making progress during training and hence a few diagnostic measures were used to help spot obvious mistakes including:

- Memory consumption
- Q-function estimate on the heldout set
- Epsilon schedule
- Average *performance runs* rewards
- Maximum absolute values of:
  - RMSProp running averages
  - Weights in each layer
  - Biases in each layer

6 Evaluation procedures

An essential part of the project and all the experiments involved is setting up training and evaluation procedures of algorithms involved. RL algorithms can generally be assessed according to three criteria - speed of learning, convergence of the Q value function and performance in completing a task by following estimated optimal policy. Both tasks - the

Grid world and Breakout - differ when it comes to the size of a state space making it impossible to evaluate their performance using the same methods.

6.1 Grid World

Small size of Grid World's state and action sets, $|\mathcal{S}| = 30, |\mathcal{A}| = 4$, allowed for exact calculation of optimal Q values for each state-action pair either by using TD or DP methods as the exact model of the environment was known. Hence a reasonable assessment method was to precompute a vector of optimal Q values and plot a graph of the L2-norm error of learnt estimtes for all possible states for a set of parameters ω - $\sum_{s\in\mathcal{S}} ||Q_{optimal}(s) - \max_a Q(s, a, \omega)||^2$. Evaluating such graphs allowed for comparing algorithms and hyperparameters with respect to their convergence rates and final *errors*. In principle it was also possible to plot the score of an agent after a test run without ϵ-greedy strategy. However, the small scale of a problem and high probability of an agent reaching the *reward tile* made it uninformative.

In most cases it may be beneficial to generate a visualisation of the optimal Q function values for all the states as shown on Figure 27. Colour of each tile corresponds to function's value where blue is the lowest and red is the highest. In general, exact values are less important than graph's characteristics due to the possibility of using different discounting values. For example, due to symmetry of the problem all tiles equidistant to the top left corner in the *Manhattan* norm should be of the same colour. Also, closer tiles should be marked by warmer and farther ones by colder colours due to discounting used.

Furthermore, it is useful to compare a learnt policy with the optimal one shown on Figure 28 by plotting the count of discrepancies between them.

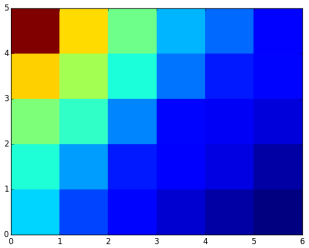

Figure 27: Q function visualisation

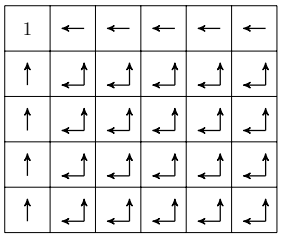

Figure 28: Grid world's optimal policy

6.2 Breakout

The Breakout problem is fundamentally different from the Grid world in a sense that it is infeasible to compute L2-norm error for each of the distinct states let alone calculate exact optimal values of a Q-function due to the size of a problem. Instead it is possible to evaluate how $\left(\max_{s\in T} \max_a Q(s, a)\right)$ changes for a set of held out states T over time as the agent gains experience. For the purpose of this report, a collection of 128 states

was obtained by saving random screenshots from a manually played game. Modification of ALE's manual agent's source code made sure that consecutive states were distinct enough by introducing an appropriate time delay between them. Also, manual control allowed to include a diverse set of samples ranging from starting positions up to high-score ones.

Another evaluation metric used in this report is the reward obtained in a *performance run* after completing each epoch of training. It is characterised by setting $\epsilon = 0.05$ [20] which effectively means that the agent tries to play optimally with a very small degree of stochasticity simulating a real game.

7 Results

Note: all the Figures included in this Section can be found in higher resolution on a CD disk attached to this report.

7.1 Grid World

Secondary task - the Grid World - served as a testbed of three Reinforcement Learning algorithms in four different hyperparameter configurations. Each method was used with both online and Experience Replay sampling as well as with both constant and linearly annealed epsilon schedules on a training run of 3000 episodes to ensure fair comparison. In case of ER, each mini-batch consisted of 15 samples drawn at random from a history of the most recent 500 observations. Each time measurement was taken under the same machine load conditions.

7.1.1 Exact Q-learning

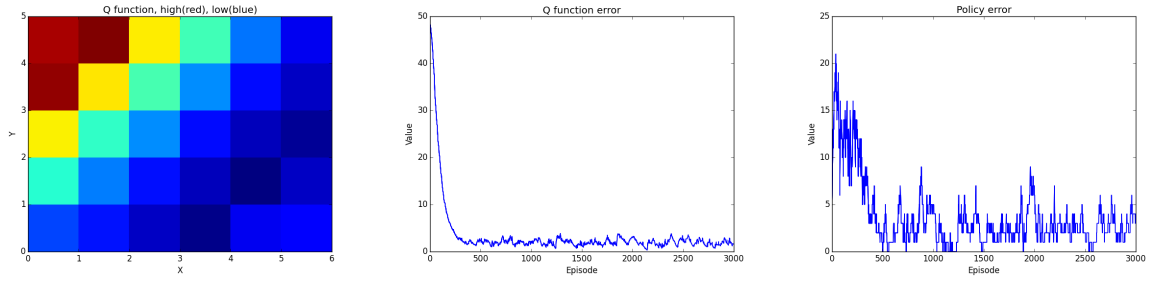

Figure 29: Exact Q-learning, online sampling, constant epsilon $\epsilon = 0.1$

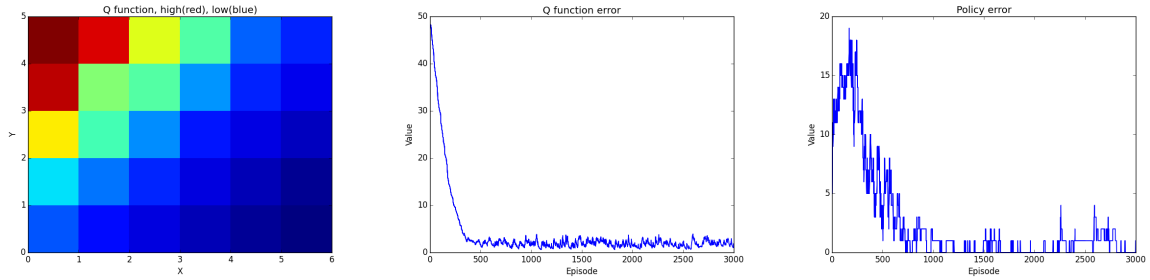

Figure 30: Exact Q-learning, online sampling, linearly annealed epsilon to $\epsilon = 0.1$

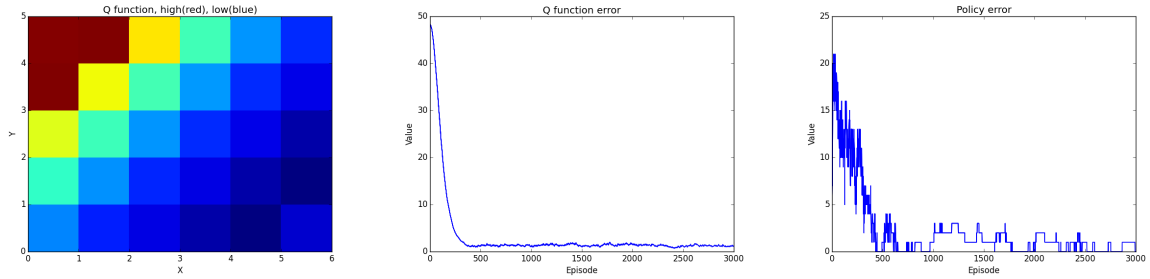

Figure 31: Exact Q-learning, ER sampling, constant epsilon $\epsilon = 0.1$

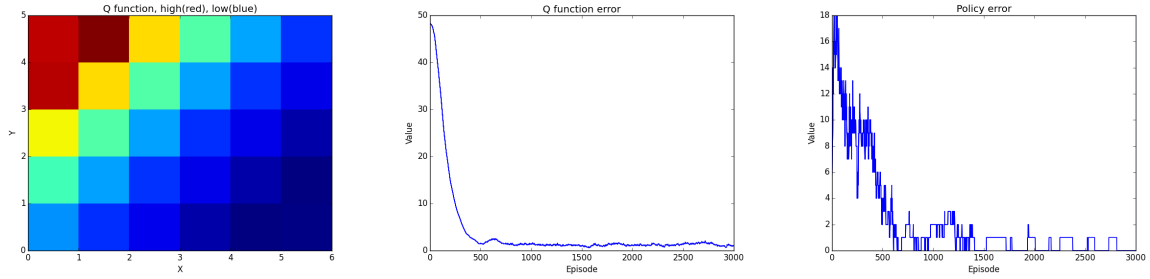

Figure 32: Exact Q-learning, ER sampling, linearly annealed epsilon to $\epsilon = 0.1$

7.1.2 Linear architectures

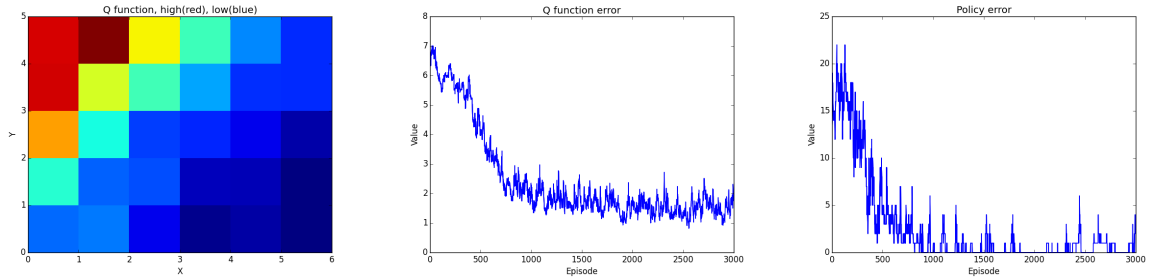

Figure 33: Linear architectures, online sampling, linearly annealed epsilon $\epsilon = 0.1$

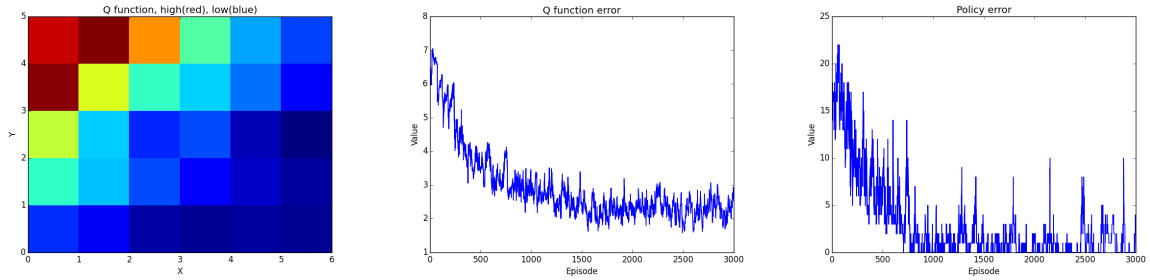

Figure 34: Linear architectures, online sampling, constant epsilon $\epsilon = 0.1$

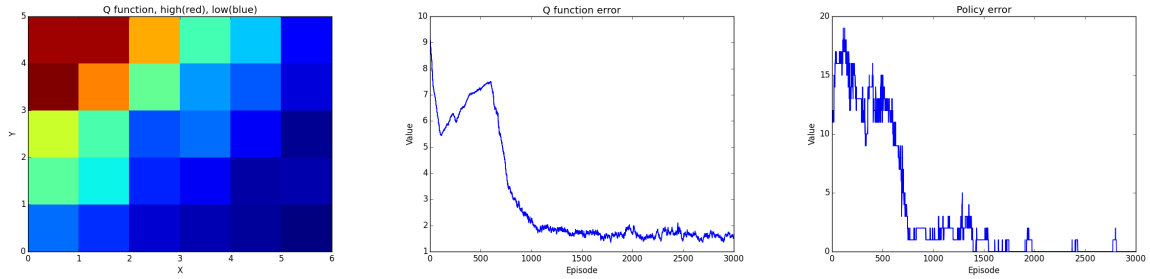

Figure 35: Linear architectures, ER sampling, linearly annealed epsilon to $\epsilon = 0.1$

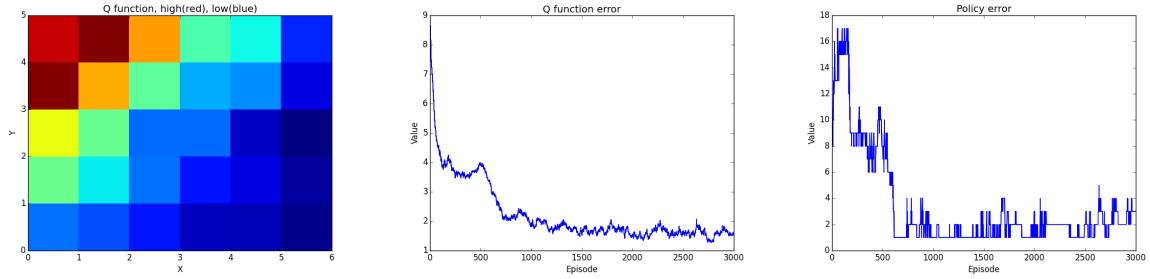

Figure 36: Linear architectures, ER sampling, constant epsilon $\epsilon = 0.1$

7.1.3 Neural network

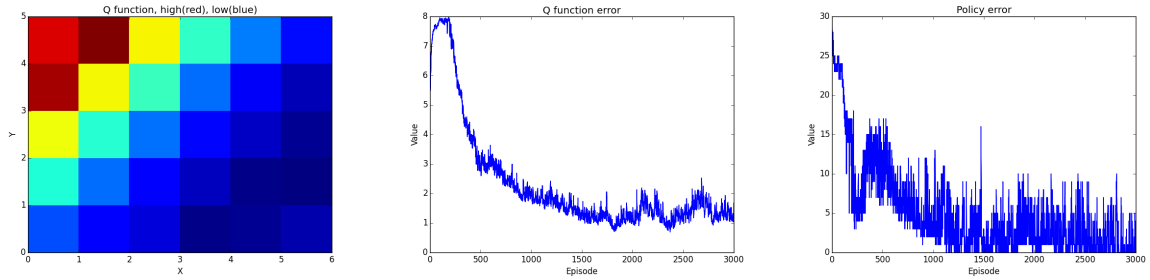

Figure 37: Neural network, online sampling, linearly annealed epsilon to $\epsilon = 0.1$

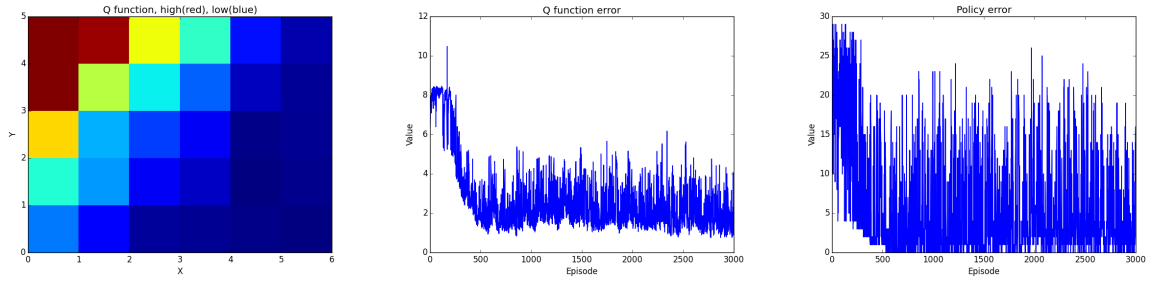

Figure 38: Neural network, online sampling, constant epsilon $\epsilon = 0.1$

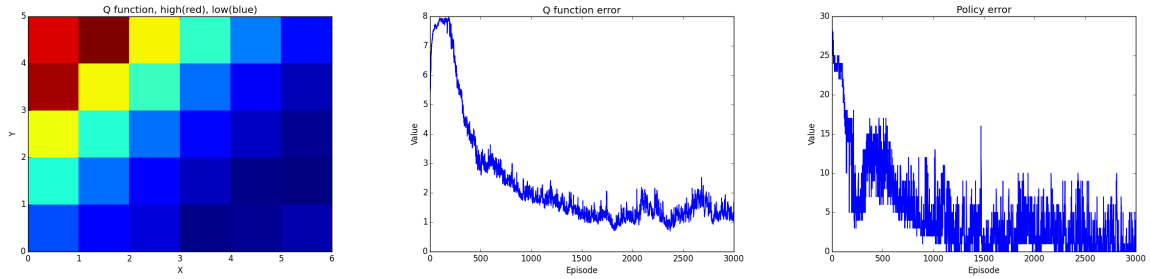

Figure 39: Neural network, ER sampling, constant epsilon $\epsilon = 0.1$

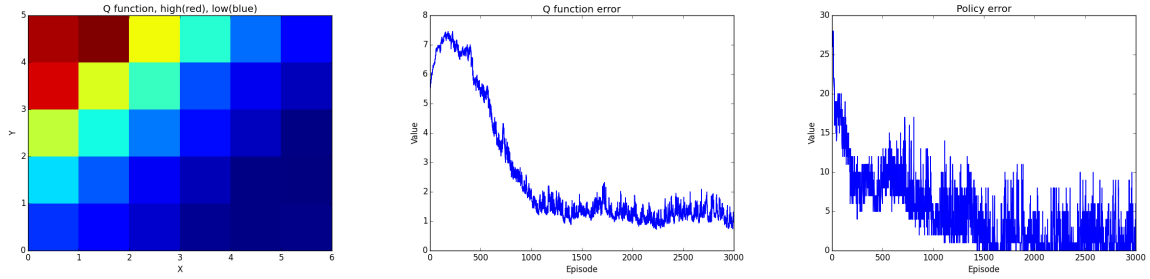

Figure 40: Neural network, ER sampling, linearly annealed epsilon to $\epsilon = 0.1$

7.1.4 Execution time comparison

	Exact Q-learning	Linear architectures	Neural network
Online sampling, constant epsilon	$23s$	$71s$	$32s$
Online sampling, linearly annealed epsilon	$21s$	$69s$	$30s$
ER sampling, constant epsilon	$36s$	$582s$	$70s$
ER sampling, linearly annealed epsilon	$37s$	$589s$	$68s$

Table 7: Algorithms' execution time data

7.2 Breakout

Graphs included in this Section can be divided into several categories depending on their role. Plots of average Q value of the manually crafted test set and average rewards collected in *performance runs* serve as main performance metrics in evaluation whether the

agent *learns*. In theory, average Q value of a heldout test set should gradually increase as the agent gains experience reaching value of around $5 - 10$ eventually. It may be used in conjunction with epsilon schedules to see what happens once the agent stops *exploring*. Average rewards were computed over multiple *performance runs* to minimize the effect of game's stochasticity.

On the other hand, maximum model parameters plots were used for debugging and spotting extraordinary behaviour which could lead to divergence. Maximum learning rates in RMSProp were also calculated and printed out after each update however their plots were not necessary to include. Other screenshots were included to point out certain algorithm features and bahaviours discussed in Section 8.

Note: although some parts of graphs included are hidden behind plot legends, initial behaviour is less important than the long term one including convergence. Also, all the plots were obtained at a time when the system ran out of memory.

7.2.1 Performance of RMSProp

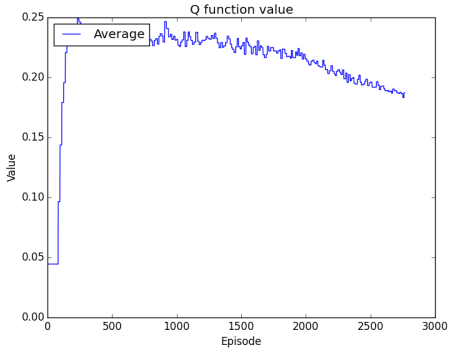

Figure 41: Q values of a heldout set

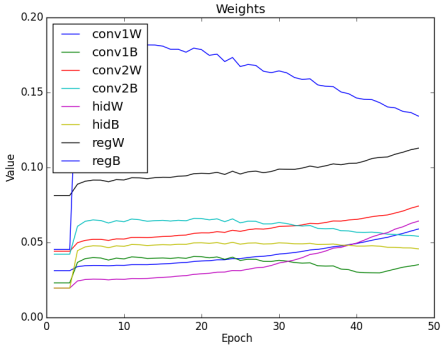

Figure 42: Maximum parameters per layer

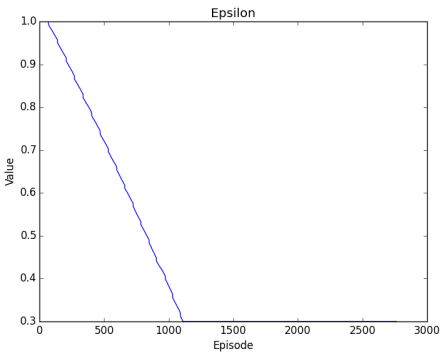

Figure 43: Epsilon schedule

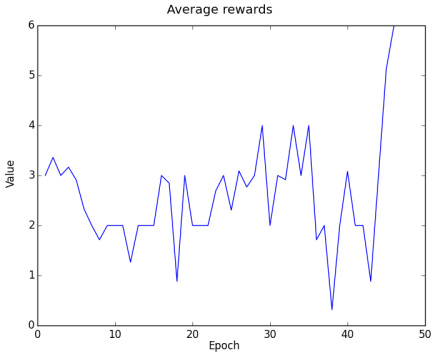

Figure 44: Average reward

7.2.2 Performance of extended RMSProp

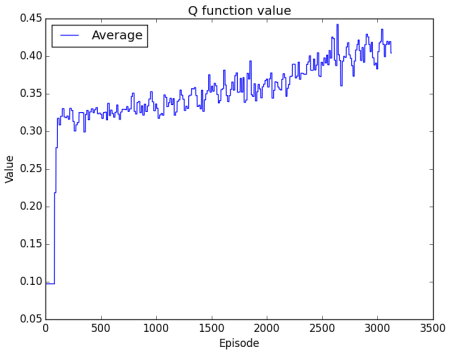

Figure 45: Q values of a heldout set

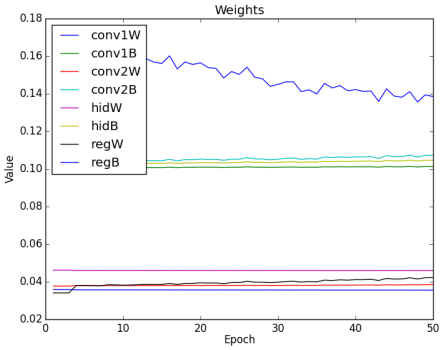

Figure 46: Maximum parameters per layer

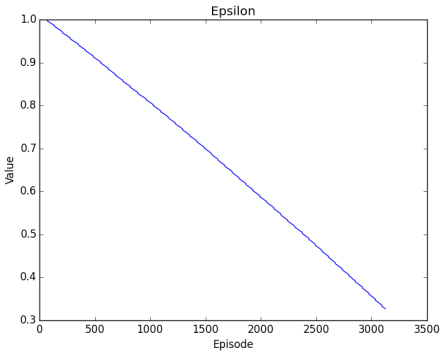

Figure 47: Epsilon schedule

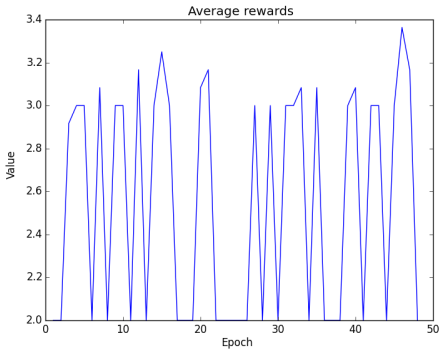

Figure 48: Average reward

7.2.3 Divergent behaviour

Tweaking the learning rate constant from $\alpha = 2 \times 10^{-4}$ to $\alpha = 3 \times 10^{-4}$ resulted in the neural network becoming divergent as shown on Figure 49. Similar behaviour was observed when $\gamma_0 = 10^{-3}$ was used instead of $\gamma_0 = 10^{-2}$.

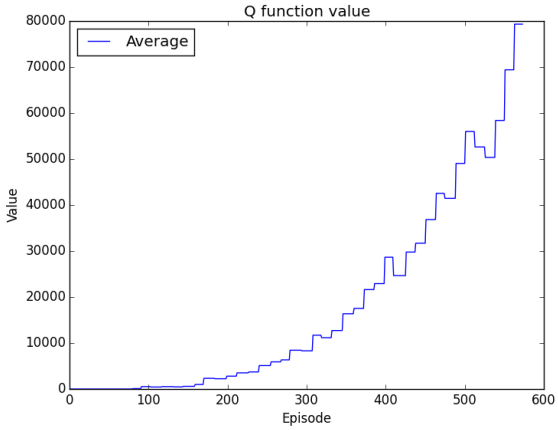

Figure 49: Neural network's divergent behaviour

7.2.4 Best performance screenshots

Screenshots were taken at the end of *best performance runs* during training.

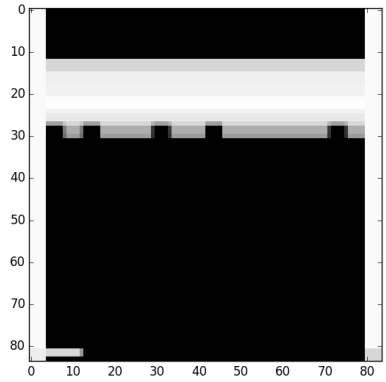

Figure 50: Best result, RMSProp

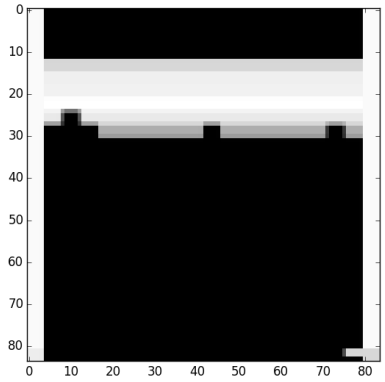

Figure 51: Best result, extended RMSProp

7.2.5 Q values propagation

Figures 52-54 show the Q value estimates of game states presented $[Q(s, a_1), Q(s, a_2), Q(s, a_3)]$, where (a_1, a_2, a_3) correspond to going left, right and not moving.

	Left	Right	Stay
Before	0.711	0.708	0.706
On impact	0.733	0.729	0.726
After	0.684	0.677	0.675

Table 8: Q value estimates

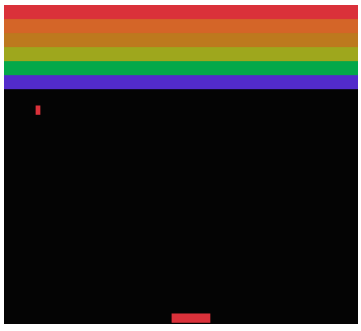

Figure 52: Before

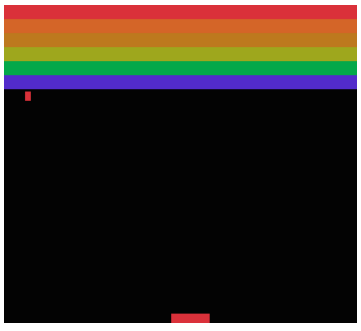

Figure 53: On impact

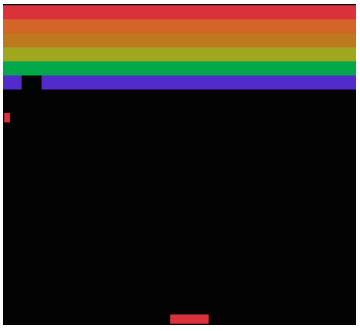

Figure 54: After

7.2.6 Convolutional kernels

Figures 55 show two example convolutional kernels *learnt* by the lowest layer discussed in Section 8.2.5.

Figure 55: Examples of 8×8 kernels learnt using MNIST Grid world representation

7.2.7 Translated trajectories

Figure 56 shows a diagram of ball's trajectories which result in similar CNN's responses discussed in Section 8.2.3

Figure 56: Translated trajectories

8 Discussion

8.1 Grid world

8.1.1 State-action convergence

Grid world is one of the simplest RL algorithms and hence it is unsurprising that all three correctly-implemented methods yielded satisfactory results. All of them managed to learn *relatively good* estimates of the value functions as well as policies quite close to the optimal one in most cases. Nevertheless there are a couple of differences worth pointing out. State-action value function error plots shown on Figures 29-32 for exact Q-learning converge noticeably faster than others and *fluctuate* considerably less as expected since each update affects just a single value estimate. On the other hand, approximate Q-learning techniques exhibited much more stochasticity due to parameter interdependence between estimates for different state-action pairs. Q function error plots tend to decrease more rapidly and converge slightly faster for linear architectures compared to a neural network although the

latter shows a *smoother* trend.

8.1.2 Policies

For small problems it is also possible to compare learnt policies with the optimal one by evaluating the number of discrepancies. Policies are generated based on value functions computed and hence in all cases policy-error plots follow similar trends to their value function counterparts both converging and featuring stochastic fluctuations. It is interesting to note that exact Q-learning results included in Section 7.1.1 show that policies converge later than the value function whereas for the two approximate methods the situation is exactly opposite. It happens that the algorithm finds *good* policies early on and uses them to improve its estimates of value functions later on.

Also, policy-error graphs for a neural network based Q-learning algorithm on Figures 37-40 tend to fluctuate significantly more than all other ones suggesting that each parameter update affects state-action value estimates of many pairs. In fact, a single feedforward neural network is used to obtain all action-value estimates of a given state and hence there is a lot of interdependence between different pairs. Also, all the model parameters carry the same weight in computations for any input decreasing the level of *adaptation* of individual neurons. On the other hand, linear architectures model uses separate sets of parameters for each action and *distance dependent* RBFs lowering the effect of a single parameter update on other state-action pairs. In theory each stochastic gradient iteration updates all the model parameters however the use of fast-decaying RBFs limits its impact to just a few tiles neighbouring in space.

It is also worth mentioning that using a linearly annealed value of epsilon usually resulted in *better* policies learnt as indicated by less *stochasticity* at convergence due to an extensive initial *exploration*.

8.1.3 Q visualisations

Colourful visualisations of optimal state-action values offered a good diagnostic measure while debugging the code as well as a good qualitative indicator of the converged parameters. A glance at the colour distribution across the Grid allows for a quick check whether obtained estimates satisfy requirements laid out in Section 6.1. As it turned out, both exact and neural network approximations produced plots of similar quality whereas linear architecture ones were considerably worse. It is especially surprising considering how *varying* corresponding policy errors were for the neural network case.

8.1.4 The effect of Experience Replay and Online sampling

Choice of the sampling strategy also has an impact on the behaviour of algorithms considered. In general it did not affect convergence rates as in most cases the agent reached its *equilibrium* state at a similar time. However, graphs obtained using online sampling

tend to show a higher degree of variability since a single update has a considerably greater effect. ER, however, *smoothed* the state-action value's error as combining samples of, on average, different characteristics made changes in the Q function less drastic. It does not make learning much faster as expected from the Breakout problem, as states are less correlated between one another.

ER's characteristic behaviour can be observed on Figures 35-36 where the error initially increased and then rapidly decreased to the converged value. These *bumps* can be explained by the fact that the algorithm most likely used *not-so-great* samples from its playing experience before it found a better trajectory through the state space and converged.

8.1.5 Parameter counts

Comparison of the two approximate Q-learning models requires taking into account both their approximating capacities and numbers of parameters used. Fixing properties of basis functions for a linear architectures model led to the total count of $20 \times 4 = 80$ parameters compared to neural network's 54 including weights and biases. In theory, a neural network is a *universal approximator* [11] and it does quite well compared to other models given smaller number of parameters especially in Q value function visualisations. It also has a slight advantage in that it does not require any *problem-specific* tailoring compared to space-distributed RBFs of linear architectures and may be treated as a black-box. In fact, it may be useful to use a popular neural networks technique, *dropout* [17], for the Grid world in future work. In principle, it would *encourage* certain neurons to specialise in a similar way RBFs make some parameters more important than others by using fast-decaying Cartesian-distance-dependent basis functions.

8.1.6 Computational expense

Evaluation of algorithms does not only require comparing their performance but also computational and storage expenses associated. Neural network based approach, even though it uses a single model for all actions, requires a whole feedforward pass to obtain value estimates for a given state resulting in 16 multiplications and 16 additions whereas a linear architectures model needs 20 sums and 20 RBF evaluations which are considerably more expensive. The difference between algorithms is reflected in computation time measurements where linear architectures are considerably slower than a neural network using respective hyperparameter settings. Exact Q-learning is the fastest one since each update affects just a single parameter and it does not use gradient descent.

On the other hand, Experience Replay changes storage requirements substantially as it requires a history of recent samples to be stored although less frequent updates may result in a computational advantage due to fewer passes both through a neural network and a linear architecture model. Using an *average* least squares cost over a mini-batch is espe-

cially useful when much larger models are used reducing the computational workload.

It is interesting to note that using ER with linear architectures resulted in a much greater performance penalty compared to neural networks not only because basis functions are more expensive to compute but also because neural network libraries are optimized towards using mini-batches.

8.2 Breakout

8.2.1 Difficulties in training

The scale of the Breakout task makes it considerably hard to train the network and obtain parameters yielding good performance. The crucial part of training is the initial exploration with high epsilon value allowing for many random actions to be taken. On average, it leads to collecting samples from many stages of a game allowing their values to propagate later during the exploitation stage. Despite that, the algorithm often ended up learning a suboptimal policy of staying in the corner as shown on Figure 57. Such strategy gives a reward of around 4-5 since at each restart the ball is thrown at 45° angles either from the centre or one of the two sides. Once the epsilon reaches its final value of 0.1, the algorithm simply keeps playing in the same way with small amount of stochasticity without much improvement in performance. Such situation might be caused by ending up in a bad local minimum of a cost function, slow learning or too short initial *exploration* phase.

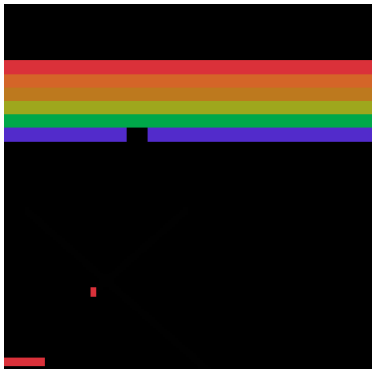

Figure 57: Bad local minimum policy learnt

8.2.2 Two CNN architectures

Results obtained towards the end of the project and memory problems encountered suggested using a smaller network which would include fewer parameters and hence require shorter training. Reducing the number of hidden units by half in the hidden layer and removing one convolutional layer corresponded to a significant decrease in complexity as the majority of parameters was actually located in the fully-connected layer. The modification improved the performance of the system considerably as it began to show signs of *learning* despite limited training time.

8.2.3 Training routine

Lengthy training of the overall system is crucial both from the Reinforcement Learning as well as from convolutional neural network's perspective. It needs to allow the agent to explore the state-space as well as obtain reasonable parameter estimates. Although it is impossible for the agent to visit every possible state, the main idea behind it is based on the notion of *state similarity*. Even though the agent may not have visited a particular state, there is a chance it has visited a *similar* one before and hence it may react *reasonably*. Convolutional layers provide invariance to feature translation aiding evaluation of states' *similarity* as, for example, a stationary paddle may bounce back balls following several *translated* trajectories as shown in Section 7.2.7 which are *similar* in a convolutional neural network sense.

8.2.4 Sensitivity to the learning rate

One of the issues related to agent's training was the choice of hyperparameters and learning rate scheme. Similarly to other Machine Learning algorithms, the training procedure was very sensitive to the value of a learning rate constant in stochastic gradient updates. Given the size of a model, it was especially important to select a learning scheme which would result in faster convergence given that observed rewards were very sparse and had to be *propagated* throughout the state space. Several trial-and-error attempts at selecting constants, namely the learning rate α and γ_0 used in RMSProp, led to network's divergent behaviour. It is especially clear in Section 7.2.5 which presents Q value estimates around the first brick scored in a game of a total score of 6. It is clear that the effects of the first *hit* have propagated to preceding states exhibiting smooth increase due to discounting however rewards obtained later in the game have not yet. In principle, Q value indicates an expected discounted return *starting* from a given state hence one should expect preceding states to have higher estimates given that the agent was capable of scoring 6 bricks. It suggests that longer training or a more efficient learning rate scheme would improve the situation.

Unfortunately due to the lack of resources, memory leaks and limited time it was impossible to compare the performance of adding a *momentum* term or of using different *adaptive* learning schemes including *Adadelta* [22] and *Adam* [21]. RMSProp and its extension described in Sections 4.3.8 and 4.3.9 provided similar performance although there could have been other differences observed between them if the entire training procedure had been completed. The extended RMSProp did not deliver significant performance gains to justify a 40% increase in the time taken to perform an update and a 50% increase in the number of model parameters. Also, more computational resources and time available would allow for better tuning of learning schedule parameters used as well as a more thorough analysis of individual neural network design decisions.

8.2.5 Kernels

One of the more interesting aspects of analysing neural networks tailored for images is a glance at the optimized convolutional kernels which correspond to features the network has learnt to react to. For large datasets of real world pictures [7] they usually represent a hierarchy of features ranging from low-level edges up to more complicated structures. Hence it would be interesting to note the particular representations the agent found most useful to play Breakout. Unfortunately due to incomplete training, robustly estimated filters were only found for the 2D-MNIST-extended Grid world probem. Despite the fact that the network managed to *approximate* the Q-function well, its complexity was too large for such a simple task leading to parameter *overfitting* characterised by seemingly *noisy* filters learnt.

8.2.6 Exploration - exploitation strategy

Unsatisfactory learning performance of the algorithm discussed above despite incomplete training procedure suggests that some modifications should be made. Given more time and resources, it would be interesting to extend the *exploration* stage allowing for more frequent visits at high-score states hence possibly aiding *propagation* of reward signals through the state space. Also, samples used in stochastic descent could be drawn from a non-uniform distribution selecting states *closer* to a reward more often [30].

9 Conclusions

9.1 The importance of the Grid World task

The introduction of the secondary problem of smaller scale, namely the Grid world, and its variations proved to be critical to the development and progress made throughout the entire project. It allowed for easier understanding of the Reinforcement Learning framework, algorithms and challenges involved in solving RL tasks. In conjunction with careful planning and modular software design it was possible to apply the same algorithms to both problems. Performance evaluation on a smaller task gave extra confidence in their correctness before running a lengthier training procedure for Breakout. In the end, the original plan was modified to include the Grid world adapted for 2D input in order to aid debugging of the convolutional neural network.

9.2 Complexity of the project

Broad scope of the project and an enormous computational expense of training model parameters on the Breakout problem turned out to cause the majority of obstacles encountered over the course of the project. After all, object oriented modular approach to designing software and careful planning allowed for graceful handling of gradually increasing complexity. Nevertheless, several iterations of most critical components were needed in order to yield satisfactory computational performance. In fact, the hardest issue to over-

come arose from using an external library *Theano* where the main memory leak was located.

It is also important to mention the importance of using a suitable type of high-performance computing facilities geared towards models used in the project. It would have been infeasible to train the algorithm on a system with a powerful CPU chip which is the case for computing facilities available in the DPO. Highly parallel nature of a neural network as well as countless transformations of 2D data are especially suitable for processing on a GPU offering a *roughly* tenfold speedup. Nevertheless, using a single computing unit proved to be a bottleneck given a lengthy training scheme needed preventing any fine tuning of *hyperparameters*.

9.3 Future work

In addition to optimizing performance and fixing memory leaks needed for successful training of a deep neural network model, methods used in this project can be modified in a number of ways possibly improving both learning and computational performance. These include but are not limited to different neural network architectures, initialization and optimization schemes, agent's strategies as well as front-end input preprocessing.

In addition, it would be interesting to apply the same model to a real-world task where signals of rewards and states from the environment are much noisier. A lot of simple problems can be controlled using a small set of actions, *represented* by a visual feed and solved relatively easily using a different state form. A good example of such is a double inverted pendulum problem [25] where the state could be obtained from a live video feed similar to Breakout.

A Risk assessment retrospective

Hazards related to extensive computer usage while working on the project were correctly anticipated and hence were successfully avoided. Striking a right balance between work and time off, including frequent breakes for quick physical exercises, was crucial for avoiding injuries.

References

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

[2] Sutton, Richard S. and Barto, Andrew G., *Introduction to Reinforcement Learning.* Figure 3.1, page 52, MIT Press,Cambridge, MA, USA, 1998

[3] Sutton, Richard S. and Barto, Andrew G., *Introduction to Reinforcement Learning.* Figure 4.6, page 105, MIT Press,Cambridge, MA, USA, 1998

[4] Lagoudakis Michail G., Parr Ronald, *Least-Squares Policy Iteration*, The Journal of Machine Learning Research, 2003

[5] DeepMind Technologies, *Playing Atari with Deep Reinforcement Learning*, NIPS Deep Learning Workshop, 2013

[6] Tieleman, T. and Hinton, G., *Coursera: Neural Networks for Machine Learning*, Lecture 6.5 - rmsprop, 2012

[7] Krizhevsky, A., Sutskever, I., Hinton, G., *ImageNet Classification with Deep Convolutional Neural Networks*, NIPS, 2012

[8] Lagoudakis, M. G., Parr, R., *Model-Free Least Squares Policy Iteration*, NIPS, 2002

[9] LBrown, L., *Accelerate Machine Learning with the cuDNN Deep Neural Network Library*, http://devblogs.nvidia.com/parallelforall/accelerate-machine-learning-cudnn-deep-neural-network-library/

[10] Lin,L.-J., and Mitchell, T., *Self-improving reactive agents based on reinforcement learning, planning and teaching*, Machine Learning, 1992

[11] Balázs, C., C., *Approximation with Artificial Neural Network*, Faculty of Sciences; Eötvös Loránd University, Hungary, 2001

[12] Sutskever, I., Martens, J., Dahl, G., Hinton, G., *On the importance of initialization and momentum in deep learning*, JMLR, 2013

[13] Hinton, G. E., Osindero, S. and Teh, Y., *A fast learning algorithm for deep belief nets.*, Neural Computation 18, 2006

[14] Bellemare, M., G., *The Arcade Learning Environment*, http://www.arcadelearningenvironment.org

[15] Bellemare, M., G., *Arcade-Learning-Environment*, https://github.com/mgbellemare/Arcade-Learning-Environment

[16] Barr, J., *Build 3D Streaming Applications with EC2?s New G2 Instance Type*, https://aws.amazon.com/blogs/aws/build-3d-streaming-applications-with-ec2s-new-g2-instance-type/

[17] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, Journal of Machine Learning Research 15, 2014

[18] Ting-Ho Lo, J., Gui, Y., Peng, Y., *Overcoming the Local-Minimum Problem in Training Multilayer Perceptrons with the NRAE Training Method*, Advances in Neural Networks ? ISNN, 2012

[19] Dieleman, S., *Even faster convolutions in Theano using FFTs*, http://benanne.github.io/2014/05/12/fft-convolutions-in-theano.html

[20] Bellemare, M., G., Naddaf, Y., Veness, J., Bowling, M., *The arcade learning environment: An evaluation platform for general agents.*, Journal of Artificial Intelligence Research, 2013

[21] Kingma, D., P., Bai Lei, J., *ADAM: a method for stochastic optimization*, ICLR, 2015

[22] Zeiler, M., D., *ADADELTA: an adaptive learning rate method*, arXiv preprint, 2012

[23] Bengio, Y.,Glorot, X., *Understanding the difficulty of training deep feedforward neuralnetworks*, AISTATS,2010

[24] Bengio, Y., *Deep Learning of Representations: Looking Forward*, SLSP, 2013

[25] Deisenroth, M., P., Rasmussen, C., E., *Efficient Reinforcement Learning for Motor Control*, 10th International PhD Workshop on Systems and Control, Hluboka nad Vltavou, Czech Republic, 2009

[26] LeCun, Y., Cortes, C., Burges, C., J., C., *The MNIST database of handwritten digits*, http://yann.lecun.com/exdb/mnist/

[27] Theano developers community, *Theano-dev Google Groups*, https://groups.google.com/forum/#!forum/theano-dev

[28] Bergeron, A., *gpuarray documentation*, http://deeplearning.net/software/ /libgpuarray/installation.html

[29] Harris, D., *On Reddit, Geoff Hinton talks Google and future of deep learning*, https://gigaom.com/2014/11/14/on-reddit-geoff-hinton-talks-google-and-future-of-deep-learning/

[30] Moore, A., W., Atkeson, C. G., *Prioritized Sweeping: Reinforcement Learning with Less Data and Less Real Time*, Machine Learning 13 103–130, 1993

[31] Sutton, R., S., *Temporal credit assignment in reinforcement learning*, Ph.D. dissertation, Department of Computer Science, University of Massachusetts, Amherst, MA 01003. Published as COINS Technical Report 84-2, 1984