University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Jakub Sygnowski
Student number: 319394

Learning agents to play Atari games using the RAM memory

Master's Thesis
in INFORMATICS

Supervisor
dr hab. Henryk Michalewski
Institute of Mathematics

November 2016

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data

Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data

Podpis autora (autorów) pracy

Streszczenie

Niedawne ulepszenia w metodach treningu i powstanie nowych rodzajów sieci neuronowych doprowadziło do stworzenia nowych algorytmów uczenia ze wzmacnianiem. Popularnym zadaniem testującym możliwości tych algorytmów są gry Atari, symulowane na współczesnych komputerach. W naszej pracy adaptujemy popularny i skuteczny algorytm deep Q-learning, który w oryginalnej wersji używa konwolucyjnych sieci neuronowych do przypadku, gdzie wejściem nie jest ekran gry, ale stan pamięci RAM maszyny Atari. Prezentujemy implementację naszej metody i wyniki ewaluacji jej kilku wersji na wybranych grach Atari.

Słowa kluczowe

Atari, deep Q-learning, pamięć RAM

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.4 Sztuczna inteligencja

Klasyfikacja tematyczna

Metodologie Obliczeniowe
Uczenie maszynowe
Algorytmy uczenia maszynowego
Programowanie dynamiczne w procesach decyzyjnych Markowa
Q-learning

Tytuł pracy w języku polskim

Uczenie agentów grających w gry Atari na podstawie pamięci RAM

Abstract

Recent advances in the methods of training and invention of the new kinds of neural networks led to creation of the new reinforcement learning algorithms. Popular framework for testing these algorithms are Atari games, simulated on modern computers. In our work we adapt a popular and efficient deep Q-learning algorithm, which in its original version uses convolutional neural networks, to the case when it is given not the game screen, but the RAM state of the Atari machine. We present implementation of our method and results of the evaluation of its few versions on the chosen Atari games.

Keywords

Atari, deep Q-learning, RAM

Subject area (codes from Socrates-Erasmus)

11.4 Artificial Intelligence

Topic classification

Computing Methodologies
Machine learning
Machine learning algorithms
Dynamic programming for Markov decision processes
Q-learning

Contents

Introduction

From the very beginning of computing, humanity is vividly interested in simulating the thought process of a mind through the field of artificial intelligence. The advances in theoretical aspects of computer science, as well as methods of manufacturing faster computers made the speed of progress of AI methods ever increasing. One can see it in many domains—from self-driving cars, through machine translation, to speech synthesis. A particularly neat area for testing intelligent algorithms is playing games.

Games are often used as a benchmark of the possibilities of AI, because on one hand, they have a well-defined evaluation metric and are cheap to simulate (which is not the case e.g. with medical diagnosis or steering robots) and, on the other hand, they offer limitless variants and levels of difficulty. While some of them (like chess, Go) require from playing agents quick search and accurate state evaluation [1], others (e.g. Doom) need reflex and good object recognition, and some (e.g. Morpion) enjoy theorem-proving methods and being treated with linear programming [2].

In this work, we focus on games published for the Atari 2600—a game console from the previous century. While these are a very small subset of all games[1], their variability offers a group of challenges. In opposition to the current trend [3, 4, 5], we train agents to play the games, seeing the RAM input instead of the screen. This problem is scarcely considered in literature (we are aware only of a poorly performing RAM agent in [6]); most of the methods described here are based on our previous work [7], presented on Computer Games Workshop during International Joint Conference in Artificial Intelligence 2016 in New York.

In this thesis, we make a couple of contributions toward making better RAM-based Atari playing agents. First, we review the foundations of the models we will be using in chapter 1. They contain both ground work on reinforcement learning from 80s and the evolving since 2012 methods of building and training neural networks.

Second, in chapter 2, we define Deep Q-learning—the algorithm we use to train playing agents. Chapter 3 consists of modification to original Deep Q-learning we tried to make it work for RAM input. In chapter 4 we show and discuss the results of evaluation of the trained methods. Chapter 5 contains ideas for further extending this work. In appendix A we describe the technical difficulties we overcame to feasibly train the models on the GPU cluster of our University's Interdisciplinary Centre for Mathematical and Computational Modelling.

Acknowledgements

First of all, I would like to thank my advisor, dr hab. Henryk Michalewski, for spending numerous hours to complete the project. He not only offered interesting scientific discussions, but also the help on the technical side.

[1] There are around 470 games produced for Atari

I would also like to thank Marc G. Bellamare for suggesting the topic of the thesis and Deepsense.io for supporting our attendance at the IJCAI conference.

The computation performed within this project were carried out with the support of grant GG63-11 awarded by the Interdisciplinary Centre for Mathematical and Computational Modelling (ICM) University of Warsaw.

Chapter 1

Foundations

Most of the groundwork of the methods we will be using was invented in the second half of the 20th century. They include reinforcement learning—a general framework defining the aim of the playing algorithm, Q-learning—a simple algorithm to learn to play a game, and basics of neural networks—the statistical model that was used to overcome Q-learning's limitations. This foundation work, joined with the recent (after 2010) advances in the methods of training and architectures of neural networks allowed to vastly improve the former results.

This chapter discusses these methods, as well as describes the Atari machine, as the games we are interested in playing are created for this platform.

1.1. Reinforcement learning

To create an algorithm learning to play a game or solve any other problem, we first have to formally define what the problem is. We will model Atari games within the framework called reinforcement learning. The main property distinguishing reinforcement learning problems from supervised learning (prediction) and unsupervised learning (clustering) is presence of two separate entities: the *environment* and the *agent*.

The environment is the physics or the rules of the game. It presents state, which can be any description of the game, to the agent, scores its action and provides him with the following state. The agent is the algorithm we prepare. It receives a state it is in from the environment, decides which action to choose there and receives the appropriate reward. Every move happens in a discrete moments of time.

The aim of the creator of the reinforcement learning algorithm is to invent a way to map the game state for each time t: s_t to the action a_t (possibly storing some inner state), so that the sum:

$$\sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{1.1}$$

is maximized. Number r_T is the reward received after doing action a_T in state s_T. The exponential averaging is called a *discounted* sum, and $0 < \gamma \leq 1$ called a *discount factor* corresponds to the level of comfort we have with receiving the awards not now, but in the future. This resembles the way people evaluate their gains—if one is promised a constant amount of money, he'd prefer to receive it rather earlier than later. We assume that every game eventually will find itself in a *terminal* state, which always transitions to a terminal state and gives reward of 0. One such progression from the start of the game to reaching a terminal state is called an *episode*.

Both agent and environment are not bound to make their decisions deterministically—in fact, it may be favorable for the agent to play randomly to some extent. In the stochastic case, the aim of the agent is to maximize the discounted sum of expected value of the rewards.

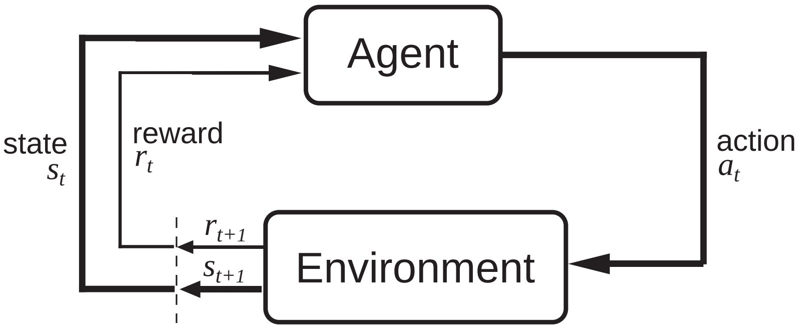


Figure 1.1: The interaction between the agent and the environment, from [8, Chapter 3.1].

The reinforcement learning, as defined above is a very general framework which can describe a broad range of problems. To make it easier to model it using statistical tools, we assume all the problems we consider are of the form of Markov Decision Process.

Markov Decision Process (MDP) [8, Chapter 3.6] is a reinforcement learning problem where a distribution of the states and the rewards satisfy Markov property.

Definition 1. Markov property A sequence of random variables x_i satisfies Markov property if and only if the distribution of each variable x_i depends only on the value of the previous variable for all i:

$$\forall_i \, p(x_i|x_{i-1}, \dots, x_0) = p(x_i|x_{i-1}) \tag{1.2}$$

In the case of the MDP, Markov property asserts that the distribution of the next states s_{t+1} and rewards r_t depends only on the previous state s_t and the action a_t:

$$\forall_t \, p(s_{t+1}, r_t|s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0, a_0) = p(s_{t+1}, r_t|s_t, a_t) \tag{1.3}$$

This simplifying assumption can be summarized as "agent has all the information he needs for making actions encoded in the state". One should note that the representation of the state and not the inner mechanics of environment, is crucial here—imagine a chess player, which sees only bottom half of the board. Even though the game is completely deterministic (assuming deterministic strategy of agent and environment, choosing opponents' moves), agent cannot reliably predict what will be the next state, as there may be pieces on the other part of the board he cannot see yet.

In the case of Atari games played based on the RAM state, the MDP assumptions are satisfied. The code of the game (saved on ROM), together with a random seed for the game define a deterministic function transforming the RAM (the game state) and awarding rewards.

It is worth noting that the algorithms based on Markov property (TD-learning, Q-learning, Markov Chain Monte Carlo [8, Chapters 6.1, 6.5 and 5, respectively]) are often used despite the violation of this assumption by the environment. The intuition behind that when the state contains enough information for agent to reasonably approximate the next state and reward distribution, it doesn't matter the approximation will not be accurate. One example of this fact is when we train agents to play Atari games based on the screen state [3]—the player moves influence the inner state (RAM) of the machine, but the changes may not be immediately apparent on the screen. Still, the game screen gives a lot of information about the game state needed to make an action, thus agent can fairly skillfully learn to make good decisions.

1.2. Q-learning

Let's call the function (distribution) mapping the current state s to the action a chosen by the agent *a strategy* (or *policy*) π.

Q-learning [9, 10] is an algorithm able to choose a strategy in an MDP. It is based on the notion of Q-value: a function mapping the state-action pair (s, a), for a fixed strategy π, to the expected discounted reward after being in the state s, making an action a and following the strategy π until the end of the episode.

$$Q^{\pi}(s_t, a_t) = \mathop{\mathbb{E}}_{r\sim p(r_t|s_t,a_t)} r + \sum_{k=t+1}^{\infty} \gamma^{k-t} \mathop{\mathbb{E}}_{r\sim p(r_k|s_{k-1},a_{k-1}),a_k\sim\pi(s_k)} r \tag{1.4}$$

where s_k is a random variable describing distribution of the states in time k, dependent of the previous state and action, as in equation (1). From here on, though, we will assume that choice of the next states and rewards by the environment as well as of actions by the agent are done deterministically—it will simplify the notation and all the statements will hold for the stochastic case when wrapped with the expectation signs. Assuming we store all the state transitions as $next\_state[s, a]$ and the rewards as $reward[s, a]$,[1] the Q-learning algorithm can be written as presented in algorithm 1.

```
for all state-action pairs(s, a) do
    $Q[s, a] = 0$ // initialize Q-values of all state-action pairs
end
for all states s do
    $P[s] = random\_action()$ // initialize strategy
end
while not converged do
    for all states s do
        $P[s] = \operatorname{argmax}_a(R[s, a] + \gamma \max_b(Q[next\_state[s, a], b]))$
    end
    for all state-action pairs(s, a) do
        $Q[s, a] = \alpha(R[s, a] + \gamma \max_b Q[next\_state[s, a], b]) + (1 - \alpha)Q[s, a]$
    end
end
```

Algorithm 1: Pseudocode of Q-learning.

The algorithm makes use of the following property:

Theorem 1. *The strategy π is optimal (maximizes the expected discounted reward) if and only if its Q-values satisfy:*

$$Q^{\pi}(s_t, a_t) = r_t + \gamma \max_b Q^{\pi}(s_{t+1}, b) \tag{1.5}$$

for all state-action pairs.

[1] In the non-deterministic case these can be approximated by, respectively, observed distribution of the next states and the sample averages of rewards in each state-action pair.

Q-learning is a representative of a general domain of algorithms called Generalized Policy Iteration (see [8, Chapter 4.6.]). Their approach is to take turns between updating the estimation of the value of the states (in our case Q-values) and updating the strategy to take into account new state value estimation.

In the case of Q-learning, the chosen strategy is a greedy one, i.e. it always chooses the action maximizing the immediate reward plus discounted value of the following state:

$$\pi(s_t) = \operatorname*{argmax}_a \big(r(s_t, a) + \max_{a^*} Q^\pi(s_{t+1}, a^*)\big) \tag{1.6}$$

where $r(s, a)$ is the immediate reward after making an action a in a state s and s_{t+1} is (a convenient notation for) a next state after making the action a in the state s_t.

The update to the Q-values is being made to force more state-action pairs to satisfy the property (1.5):

$$Q_{new}(s_t, a) := \alpha Q_{old}(s_t, a) + (1 - \alpha)\big(r(s_t, a) + \max_{a^*} Q_{old}(s_{t+1}, a^*)\big) \tag{1.7}$$

α (also called step size) is a parameter of the algorithm deciding how fast it should move the Q-value estimations toward the ones locally satisfying (1.5). Bigger values lead to faster training, but the convergence proof for constant step size requires sufficiently small α [10, section 3.]. It was also shown in [11], that the Q-learning algorithm converges when the variable step size satisfies:

$$\sum_i \alpha_i = \infty, \quad \sum_i \alpha_i^2 < \infty \tag{1.8}$$

1.3. Neural networks

Many of the current advances in various domains of AI are the effect of improvements of an old[2] statistical model, inspired by how the brain works: neural network. In this section we define it formally and describe the process of finding best parameters for it.

1.3.1. Feedforward neural network

The simplest neural network, called *feedforward* neural network or *multilayer perceptron* was introduced in 60's [13]. It consists of a couple of *layers*:[3] each layer accepts some inputs, processes them, and outputs them as an input for the next layer (see figure 1.2). The first layer is called input layer, the last—output layer and all the layers in between—hidden layers.

A layer is a composed of *nodes* (also called *neurons*). A node accepts as input the previous layer's nodes' outputs, calculates a linear combination of them and applies a nonlinear function, called *activation* function to the result (see 1.3). The typical choices for the activation function are sigmoid: $\frac{1}{1+e^{-x}}$, hyperbolic tangent, and (popularized recently) rectified linear unit (ReLU): $\max(0, x)$.[4]

The trainable parameters of the model are weights of the linear combination of each of the nodes. The intuition behind introducing multilayer, multinode architecture is that different nodes will learn different statistics useful for producing the output, e.g. eye or nose detectors for face recognition. The neurons from the earlier layers will learn simple dependencies in the data (e.g. detection of edges) and the further neurons will combine their information, finding more sophisticated relations (e.g. dog detector neurons).

[2] The predecessor of today's neural network, perceptron, was invented in 1958 [12].

[3] In the current state-of-the-art models, the number of layers reaches thousands, see e.g. [14]

[4] A thorough, yet easy to understand discussion of the various activation functions can be found in [15].

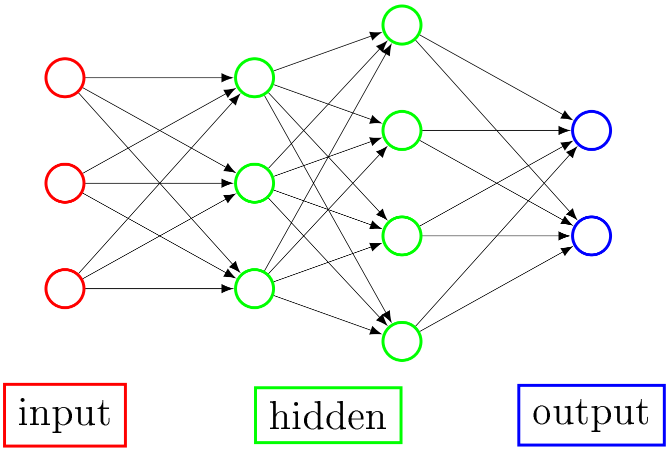


Figure 1.2: Architecture of nodes in a feedforward neural network. Each node receives the outputs of the previous layer.

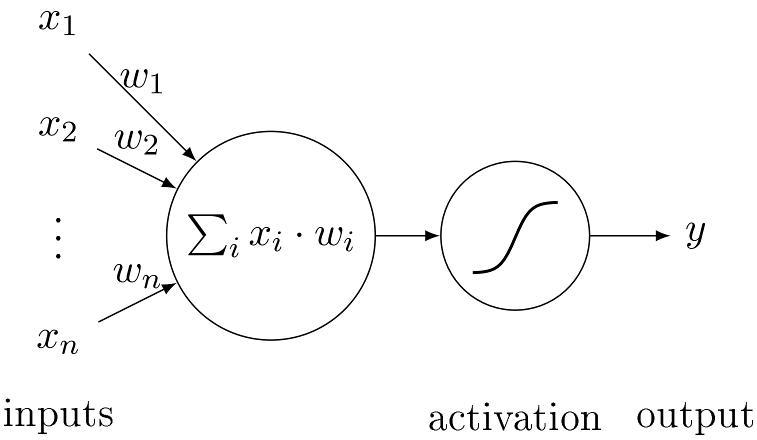


Figure 1.3: Details of a neural network node.

1.3.2. Gradient descent

There are a lot of optimization algorithms used to train neural networks, i.e. find the best set of parameters (weights) for a given network architecture and the data. All of them are a modification of an algorithm of gradient descent [16].

To use it, we first need to choose a loss function, which is a differentiable function of the output of the neural network that correlates with how well the model is doing its job; its value should be high for the bad models and low for the good models. For example, if we want to predict the depth of each pixel in a photo (useful e.g. for self-driving cars), we could use a dataset of images accompanied with their depths estimated with a 3D camera. As a loss function, we could use a mean-squared error between the real depths and those predicted by the model:

$$L = \frac{1}{\#\text{ images}} \sum_{image} \sum_{x,y} (\text{real\_depth}(image, x, y) - \text{model\_output}(image, x, y))^2$$

Then, to update the weights of the model to minimize the loss, gradient descent calculates the gradient of the loss with respect to every parameter of the model and moves the weights in the direction of the steepest descent by a small amount ε, called *step size* or *learning rate*. Choice of a bigger ε leads to faster learning, but makes the optimization prone to missing good areas of space or diverging completely. Smaller step size leads to more stable learning.

The gradients of every weight is determined with the help of chain rule:

$$\frac{\partial}{\partial x}(f \circ g) = (\frac{\partial}{\partial x} f \circ g) \cdot \frac{\partial}{\partial x} g \tag{1.9}$$

The equation (1.9) suggests an order of calculating the derivatives: we should first calculate the ones of the weights in the output layer, then previous hidden layer, and so on until the input layer. For example with squared error, sigmoid activation function, datapoints x_i, and target outputs y_i we have:

$$\begin{aligned}
\frac{\partial L}{\partial w_{\text{input},1}} &= \frac{\partial}{\partial w_{\text{input},1}} \sum_i (y_i - \text{output}(x_i))^2 = \\
&= \sum_i 2(y_i - \text{output}(x_i)) \cdot \frac{\partial}{\partial w_{\text{input},1}} (y_i - \text{sigmoid}(\sum_j w_{\text{output},j} \cdot \text{hidden}(x_i, j))) = \\
&= \underbrace{\sum_i -2(y_i - \text{output}(x_i)) \cdot \text{sigmoid'}(\sum_j w_{\text{output},j} \cdot \text{hidden}(x_i, j)) \cdot \sum_j \text{hidden}(x_i, j)}_{\sum_j \frac{\partial L}{\partial w_{\text{output},j}}} \cdot \frac{\partial}{\partial w_{\text{input},1}} \dots
\end{aligned} \tag{1.10}$$

Because the process of gradient calculation proceeds from the end of the network toward the beginning, it is called *backpropagation* or *backward pass* (in contrast to *forward* pass, when the network is evaluating outputs).

The calculation of a full loss, based on all examples can be infeasible: current datasets often consist of tens of gigabytes of data (see e.g. COCO challenge dataset: [17]) and to train a network it is often needed to make a couple of thousands of parameter updates. To speed the process up, it is common to approximate the full loss function (thus the gradients, too) by a loss calculated on a random subset of data examples. More often than not, a random subset of 100 examples will possess statistical properties indistinguishable from the whole dataset, leading to accurate (and fast to obtain) estimates of gradients. This process is called *minibatch* training (and the whole algorithm a *stochastic* gradient descent) and the size of the *batch* is another hyperparameter of the model. There is a trade-off in its choice—bigger batch size gives better gradient estimates, but takes longer to calculate.

The role of nodes in a neural network is symmetric—there's no reason a particular neuron should behave differently than its neighbor. To enforce neurons to learn various aspects of the data, we initialize the weights randomly. The experiments show that it suffices to make different neurons learn different weights—intuitively, the model with different neurons will have easier time predicting output, as it will be combining various types of information.

One can summarize this section by presenting a stochastic gradient descent pseudocode in algorithm 2. An example implementation for a toy problem can be found in file `code/gradient-descent.ipynb` on the CD.

1.4. Deep Learning

The neural networks were not often applied in modelling until recently. There was a number of factors which contributed to their current successes. In this section we describe shortly the most important of them.

```
initialize all weights randomly;
while not converged & not exceeded maximum number of iterations do
    pick a random subset $S$ of data examples;
    calculate the loss function for the examples in $S$ in forward pass;
    calculate the gradients of the loss and update each weight in a backward pass:
    $w := w - \varepsilon \cdot \frac{\partial L}{\partial w}$
end
```

Algorithm 2: Pseudocode of gradient descent.

1.4.1. Computing power

Creating good prediction or classification models requires a lot of computation power.[5] Neural networks, with their big expression power can provide good results, but only when supplied with enough data. When we pass a little data to a deep neural network, we can often see an *overfitting* effect—the model performs well on the data it saw (training data), but generalizes poorly to unseen examples (test data).

In a world with limited computation power, best results will be biased towards simpler models, like linear regression, random forest, or support vector machines. Out of the whole continuum of $R^n \to R^m$ functions, elementary models will pick a small subset of smooth, easy, reasonable ones and will pick the best one with the use of limited data. As the subset of functions that can be expressed is small, they are quite different from each other and it is easy for the model to pick the best function.

When a more involved model is trained, it can represent more functions, but at the expense that it requires more data (thus more computation power). When faced with data scarcity, it doesn't choose the best function among all that it can represent—it falsely detects noise in the data as significant patterns and erroneously treats insufficiently often occurring patterns as noise.

Fortunately, thanks to game industry, we saw a rapid increase in computation speed and memory bandwidth of graphics processing units (see figure 1.4). Most of the current state-of-the-art algorithms for finding best parameters for a neural network can be defined in terms of common matrix operations. Due to sophisticated library implementations and good parallelization properties, these operations can be computed fast using a GPU, leading to efficient training of neural networks. While we saw vast speedups moving from a high-end CPU to a middle range GPU, it is fair to mention that some of the algorithms, e.g. actor-critic methods described in [5], do not experience significant improvement in training time on a GPU in comparison to a CPU.

1.4.2. Better training algorithms

The speed of convergence of vanilla stochastic gradient descent can be improved in many cases. For example, if the optimization space is a valley with high hills and a river with a little slope in the bottom, SGD will zigzag from one wall of the valley to the other, due to higher gradients between the walls than in the river.

There are a lot of improvements of the SGD to faster move in the optimization space in the situations like this. They include momentum [19], Adam [20], or Adadelta [21]. I will describe RMSProp [22], the algorithm that we use in our code. One can read about the others

[5] Deep learning community jokes that the time to train a state-of-the-art model stays a week, regardless of the increase in the computing power.

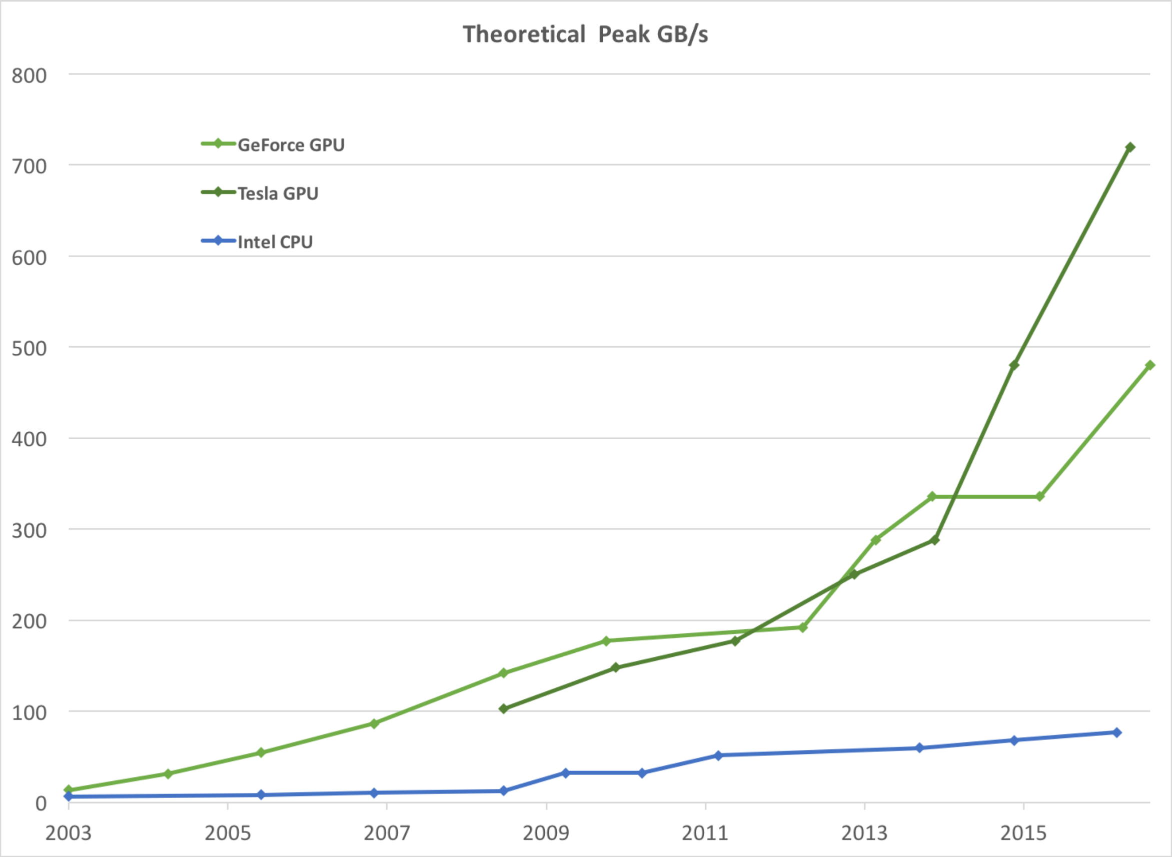


Figure 1.4: Comparison of best CPU and GPU memory bandwidths in time. Note that the memory bandwidth often constraints the speed of neural network training. Plot comes from [18].

in [23, chapter 8.3.] in detail.

RMSProp

The basic idea of RMSProp is dividing the gradient by the square root of the sum of previous gradients' squares pointwise. This way the directions which keep having high gradients in the course of training, will be scaled more and more and directions will low gradients will be more and more "preferred" to traverse along.

To prevent vanishing the learning rate too much and significantly slowing down the learning near the end, instead of calculating the sum, RMSProp stores the exponential average of the history of gradients' squares.

The pseudocode of RMSProp can be found in algorithm 3.

1.4.3. Convenient libraries

Another factor causing deep learning models to rise is the creation of open source libraries for symbolic computation. A mere 6 years ago, if a student or a scientist wanted to experiment with neural network models, he had to implement the neural network as a couple of matrix multiplications, calculate the gradients on paper and backpropagate them in code, and implement a version of gradient descent. Not to mention that performing the computation on

```
input: Initial parameters $\theta$, learning rate $\varepsilon$, decay rate $\rho$, small constant $\delta$ for
       numerical stability
Initialize gradient accumulation $r \leftarrow 0$
while not converged do
    Pick a random subset $S$ of data examples
    Calculate gradient: $g \leftarrow \nabla_\theta \sum_{(x,y) \in S} L(f(x,\theta), y)$
    Accumulate squared gradient: $r \leftarrow \rho r + (1-\rho) g \cdot g$ (pointwise multiplication)
    Compute parameter update: $\Delta\theta \leftarrow \frac{-\varepsilon}{\sqrt{\delta + r}} \cdot g$ (pointwise multiplication)
    Update parameters: $\theta \leftarrow \theta + \Delta\theta$
end
```

Algorithm 3: Pseudocode of RMSProp, adapted from [23, Chapter 8.5.2].

GPU using a direct memory manipulation like CUDA increased the code complexity twice.

Fortunately, around 2010 open source libraries, greatly simplifying the process of implementing deep learning models, started to appear. Currently there are 3 such frameworks which are popular: Theano, Torch and Tensorflow.

Theano [24] is a python library developed by Montreal Institute for Learning Algorithms, which the author made a humble contribution to.[6]

Torch [26] is a library allowing to write neural networks in Lua, used and developed by Facebook.

Tensorflow [27] is a library made by Google, supporting writing in C++ and python. It was the first to seamlessly allow the distributed training of models.

The ideas behind these libraries are similar. All of them allow the user to write the high-level code that will be interpreted as a symbolic computation forming a directed acyclic graph. Then, during execution, it is compiled to a low-level code able to run on a GPU. The libraries decide on the order of calculating the nodes in the graph (called *Ops*).

Treating the code written by the user as symbolic expressions and deferring its evaluation allow to perform low-level optimization (like in compilers—in a sense these libraries *are* compilers), symbolically calculate the gradient of an expression, and allow for different backends (CPU vs. GPU).

You can see a short example of Theano code in snippet 1.5, which is also available as `code/theano.ipynb` on the CD.

```
In [1]: import theano

        x = theano.tensor.dscalar("x")
        y = 3 * x - x * x
        dy = theano.tensor.grad(y, [x])[0] # -2x + 3
        f = theano.function(inputs=[x], outputs=[y, dy])
        f(3)

Out[1]: [array(0.0), array(-3.0)]
```

Snippet 1.5: Example Theano code

[6] The contribution can be accessed in [25] and contains possibility of evaluation just a subset of outputs of an already compiled Theano function.

Figure 1.6: Picture of the Atari machine.

Theano, Torch and Tensorflow in their basic form are libraries to express any kind of computation. The popularization of them fueled the creation of many higher level frameworks, even further simplifying implementation of deep models. These libraries, including Keras [28], blocks [29] and TFLearn [30] allow for implementing typical transformations, like an all connected layer, softmax, or popular activation functions, in one line. They make it easier to read the data, visualize the learning process, and implement the popular optimization algorithms.

1.5. Atari 2600

Atari 2600 [31] is a game console created in the late 70s. It has a processor running at around 1 MHz, mere 128 bytes of RAM memory and a joystick with 18 actions (9 positions $\times\{\text{fire}, \text{no-fire}\}$). To play a game, one needs to insert a cartridge with a ROM of a game (of size $4kB$) and connect the console to the TV screen. It was producing a new game screen with frequency 60Hz using some tricks,[7] needed to show the full image when the RAM fitted only a line of it. A picture of the Atari 2600 can be seen in figure 1.6.

The interest in Atari within AI community arose after the publication of [6]. It introduces Arcade Learning Environment, a framework which allows for easy creation and evaluation of agents playing Atari games. It encompasses Stella [33], the Atari simulator for computers. Thanks to ALE, the programmer can read the screen as a python array, know which actions are possible in each game and read rewards without the need to parse them from the screen nor understanding the details of RAM memory of each game.

Some time after the release of ALE, OpenAI, an emerging player in AI, created a convenient wrapper on top of it: OpenAI Gym [34]. It further simplifies the evaluation of the agents, introduces some new, non-Atari environments to test the agents, and make it possible to visualize the training on OpenAI page, encouraging scientists to open source their ideas.

The diversity of Atari games made it a good challenge for AI agents. There are both very simple games, like Pong, where a good initial position of the paddle can guarantee winning and very involved ones, like Montezuma's Revenge requiring to collect items and

[7] Interested reader can read a short article about "racing the beam" in [32].

use them correctly.[8] The work on creating agents capable to play many of the Atari games well resulted in many breakthroughs in AI, including winning the Go game with a human world-champion [1] and creation of deep Q-learning, to which the next chapter is dedicated.

[8] The best to our knowledge method for Montezuma's Revenge [35], requires manual formulation of the game's objects and sub-goals, what makes it hardly transferable to other environments.

Chapter 2

Deep Q-learning

Deep Q-learning[1] is a reinforcement learning algorithm introduced in [3]. It is known for being able to learn agents to play Atari games using the game screen as input. Mathematically, it is an extension of Q-learning (described in section 1.2), using neural networks as a Q-value approximator.

Even though Q-learning is a correct algorithm in a sense that it converges in the limit to the optimal strategy, the rate of this convergence is slow. It comes to no surprise—to propagate the rewards and correctly estimate Q-values of all states, each of them have to be visited multiple times.[2] With raw Atari frames of size $210 \times 160 \times 3$ (width $\times$ height $\times \{\mathrm{R}, \mathrm{G}, \mathrm{B}\}$) and color depth of 128, the number of possible input states can be estimated as:

$$\#\ \text{states} = 128^{210*160*3} \approx (2^7)^{2^{16}} \approx 2^{2^{18}} \approx 2^{256000} \tag{2.1}$$

which is way too much to store in this universe, not to mention process multiple times on a computer.

Main idea of deep Q-learning is to use some form of generalization between states. This way, seeing a reward in a given state-action pair, we will get information not only about this particular state, but about similar states as well. This will also let us to store the Q-value function in a concise form, alleviating the exponential state explosion problem outlined above.

We decide to use a neural network as an approximator of the Q-value function. As we have no supervised signal about the values of the Q function, we use the theorem 1—we know that the characteristic property of optimal strategies π is:

$$r_t + \gamma \max_b Q^\pi(s_{t+1}, b) - Q^\pi(s_t, a_t) = 0 \tag{2.2}$$

Thus, we define the loss function the network (compare section 1.3.2) as the square of the left-hand side of the above equation. We hope that by minimizing this value, the strategy will become close to the optimal one.

Definition 2. Loss function of the neural network with regards to its parameters θ_t in time t:

$$L(\theta_t) = \mathbb{E}_{s_t, a_t, s_{t+1}} \big(r_t + \gamma \max_b Q(s_{t+1}, b|\theta_{t-1}) - Q(s_t, a_t|\theta_t)\big)^2 \tag{2.3}$$

[1] known also as deep Q networks or DQN

[2] [36] presents theoretical analysis of Q-learning speed of convergence. The raw Q-learning's complexity is estimated as $O(n^3)$ in the number of states.

Let's explain notation in more detail. t is the discrete time describing changing frames of the game; it is used only to distinguish previous (s_t) and next (s_{t+1}) state as well as previous (θ_{t-1}) and current (θ_t) parameters. Q is the function mapping state-action pairs to the estimation of their value (expected discounted reward). The role of Q is fulfilled by a neural network. The part $|\theta_t$ comes from the fact that the values of Q function are be dependent on the parameters θ_t of the network. We treat previous parameters θ_{t-1} as fixed, so the only trainable parameters in L are θ_t. The estimation sign $\mathbb{E}_{s_t,a_t}$ comes from the fact we will not score all the state-action pairs, but rather the ones that appear in real games. In fact, a more appropriate notation would be to sum over the observed state-action pairs; the current one is used to simplify notation.

We can differentiate the loss function with respect to the parameters:

$$\begin{aligned}\nabla_{\theta_t} L(\theta_t) &= \nabla_{\theta_t} \mathbb{E}_{s_t,a_t,s_{t+1}} \left(r_t + \gamma \max_b Q(s_{t+1}, b|\theta_{t-1}) - Q(s_t, a_t|\theta_t)\right)^2 = \\ &= \mathbb{E}_{s_t,a_t,s_{t+1}} \nabla_{\theta_t} \left(r_t + \gamma \max_b Q(s_{t+1}, b|\theta_{t-1}) - Q(s_t, a_t|\theta_t)\right)^2 = \\ &= \mathbb{E}_{a_t,s_t,s_{t+1}} 2\Big(r_t + \gamma \max_b Q(s_{t+1}, b|\theta_{t-1}) - Q(s_t, a_t|\theta_t)\Big) \nabla_{\theta_t} Q(s_t, a_t|\theta_t) \qquad (2.4)\end{aligned}$$

Note that θ_{t-1} is treated as a constant with respect to θ_t. Even though this derivation is unnecessary to implement deep Q-learning (as it is usually done by a library), it gives some intuition about how our optimization algorithm will work.

$Q(s, a|\theta)$ will be implemented as a neural network that accepts the state of the game and returns a multidimensional vector—each coordinate will correspond to a Q-value of one action. At each time step, we will query the network with current state and make the action with the highest Q-value. Then, we will collect tuples: s_t, a_t, r_t, s_{t+1} of current state, performed action, received (immediate) reward and the next state. To use such a tuple to decrease the loss (2.3), we will calculate $\max_b Q(s_{t+1}, b)$ and $Q(s_t, a_t)$ in a forward pass through the network, and $\nabla_{\theta_t} Q(s_t, a_t)$ using backpropagation (starting from the output node corresponding to the chosen action). We will use these expressions to calculate the derivative of the loss $\nabla_{\theta_t} L(\theta_t)$ and update the parameters using RMSProp—a modification of gradient descent described in section 1.4.2.

The core algorithm, defined above, is not powerful enough to learn to successfully play Atari games using screen output. To make it work better, authors employed a couple of improvements, described below.

2.1. Preprocessing the screen

Low-detail graphics of Atari games have low information density. If we entered the screens as they was to the network, it would have to learn the dependencies between pixels itself, leading to less stable and longer training. To circumvent this problem, authors of DQN downscale the screens to gray-scale 110 × 84 images. Then, they further crop the central square of 84 × 84 pixels of the image, to be able to easily pass it as input to a convolutional network.[3]

2.2. Exploration-exploitation trade-off

When we play the game using Q-learning, we choose the actions that have the highest Q-values for the state we are in. For testing, that is a reasonable approach—we want our agent to

[3] Convolutional networks, as more appropriate to process screen than the main topic of this work—RAM, are not described in detail here. A detailed explanation of convnets can be found in chapter 9 of [23].

play as good as he could, given current state value estimation. But for training, the following scenario can happen: during our first game, say we chose action a_0 in the initial state s_0, proceeding to s_1. We received some small, but non-zero reward r and continued the game from there. When we started the second game, we were obliged to use the action a_0 in s_0, as it is associated with non-zero reward (and we have little knowledge about the state value anyway). We will be doing this choice on and on, until the value estimation of state s_1 drop, so that the small reward will be outweighed by it.

In this situation, our basically random choice at the start of training influences our behavior for a very long time, preventing us from trying other options, that may be more viable. This problem is known as exploration-exploitation trade-off; on one hand, we would like to make the best actions we know (to *exploit*), but on the other, we would like to build our knowledge about the effects of different actions (to *explore*).

One of the solution to this problem, is, instead of using a greedy strategy (choosing the actions with the best Q-values), to use so called ε-greedy strategy—to choose a random action with probability ε, and with probability $1 - \varepsilon$ the one with the highest Q-value estimation. By choosing ε we decide how much time we want our agent to spend on exploring new actions vs. exploiting the ones he knows are good.

This technique is not uniquely associated with DQN; in fact it was used with bare Q-learning and other reinforcement learning algorithms from early on.

2.2.1. ε decay

To set the exploration-exploitation trade-off in an appropriate place, the DQN inventors decided to use changing ε, namely one decaying linearly from 1 to 0.1. This way, at the beginning of training, when the state value estimations are poor, the agent does more exploration of new actions and further in the game, he chooses the best action according to the Q-values of states more often. Note that the random actions are done only in the training part—when the performance of a trained agent is evaluated, a flat, low value of $\varepsilon = 0.05$ is used.

2.3. Frameskip

In many Atari games, developers use visual illusion to make games look better. For example, in Space Invaders, in which the aim of the user is to shoot down the enemy ships and avoid their shots, the shots are not shown on the screen on every frame. Instead, the user can see the shot on one frame, then for a couple of frames the shot cannot be seen on the screen, then it appears somewhere further. For humans, the illusion of movement is good enough.

If we tried to pass the one-frame input to the neural network, it'll have problems learning the appropriate Q-values, because without seeing the shot, the network will not know if in the next state the player will be hit and will lose a life or the enemy will be killed and user will gain points. Formally speaking, the markovian assumption we were using from the very beginning will be violated.

The solution to this problem is to pass not the last frame of the game, but the last k frames (in [3], $k = 4$ is used for most of the games) to the network, choose the action based on those frames and repeat it k times.

This serves two purposes. First, it solves the problem of "blinking" objects, unless they disappear for at least k frames.

Second, it helps for learning dynamics of the model. Let's imagine the agent is in a state, such that he will receive a big reward if he presses "left" for half a second. Assuming A possible actions, 60 frames per second, and basically random actions at the beginning, this will happen with probability $A^{-(60/2)}$. If we change the action every $k = 4$ frames, the probability to make the left movement for half a second will dramatically increase to $A^{-(60/(4 \cdot 2))}$, as we will now make 4 times less decisions.

Of course, this improvement (called frameskip) limits the amount of strategies that we can learn—our agents are now only able to express strategies that change actions at least every k frames, but, unless the game need superhuman reflex (like, e.g., boxing [37]), using bigger than 1 frameskip value helps to train better agents in practice.

2.4. Experience replay

Another method used to effectively train the neural network agents used in DQN is experience replay. Training a neural network to achieve good performance can be a long process. For a regular supervised tasks, like image recognition or machine translation is it not uncommon to have to process the whole dataset multiple (tens to hundreds) of times (called epochs) to get the optimal parameters for given hyperparameters, even with the use of minibatch training.

In the reinforcement learning setting, we don't have a dataset per se, we rather interact with environment and observe its reactions. To be able to reuse the training examples multiple times, we save the history of our interactions with environment (in a form of a list of tuples, containing (state, action, reward, next state)) in the memory. During training, we do the actions according to the strategy defined by the network's Q-values estimation, but to update the weights, we choose a random batch of tuples from the memory and calculate the gradients using them.

This method allows us to look at each interaction (example) multiple times during training. It also improves statistical properties of the gradient: as we use stochastic gradient descent, we would like examples in each batch to be representative of the whole history of observations; choosing the elements of the batch randomly, we achieve better estimate of the full loss, as the consecutive observations are strongly correlated.

Due to the limits in the size of memory, the history of interactions contain last n (in the order of tens-hundred to millions), not all past observations.

Chapter 3

Deep Q-learning for RAM

In our work, we adapt DQN to learn not from the game screens, but rather from the RAM state of the Atari machine. In the following sections we describe the previous work in the domain of playing Atari games using RAM and the details of our approach, explained also in the paper [7].

3.1. Related work

In comparison to screen-based models, the models for playing Atari games based on RAM are not often studied in AI literature.

The work [38] presents a classical planing algorithm to play Atari games based on the RAM. Since the RAM contains only 128 bytes, one can efficiently search in this space. Nevertheless, the search is too slow to play in real-time—the evaluation of the method assume that one can spend as much time as it is needed to decide on a move. To the best of our knowledge, the only RAM-based agent that does not depend on search was presented in [6]; we cite these results as `ale_ram`.

As the main benchmark of our work, we use original (screen-based) DQN model, introduced in [3] and described in detail in chapter 2. While it was improved in a number of ways in [4, 39, 40, 41], leading to better architecture, hyperparameters, and Q-function estimates, these enhancements came at the cost of longer training time. Choosing a less sophisticated baseline [3] makes it possible to fit a training of a single model in roughly 48 hours using our technical setup (described in section 3.3) while still making it possible to verify feasibility of learning from RAM. We refer to this baseline architecture as `nips`.

3.2. Games

We test our models on three games:

Bowling: simulation of the game of bowling; the player aims the ball toward the pins and then steers the ball; the aim is to hit the pins [42, 43].

Breakout: the player bounces the ball with the paddle towards the layer of bricks; the task is to destroy all bricks; a brick is destroyed when the ball hits it [44, 45].

Seaquest: the player commands a submarine, which can shoot enemies and rescue divers by bringing them above the water-level; the player dies if he fails to get a diver up before the air level of submarine vanishes [46, 47].

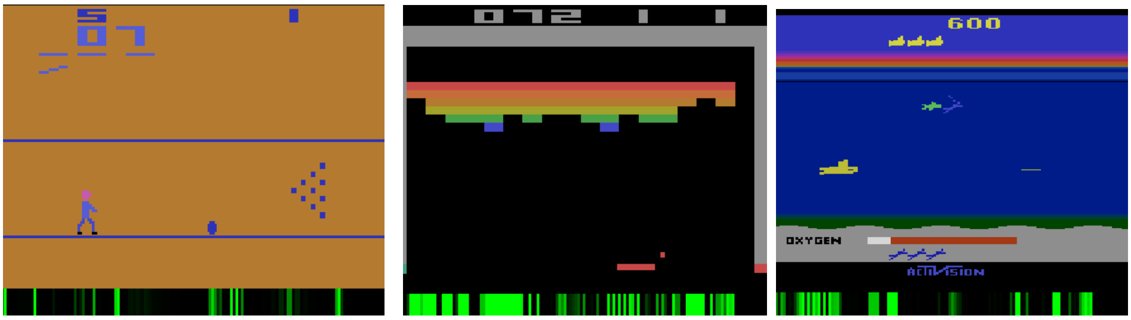


Figure 3.1: From left to right Bowling, Breakout and Seaquest. The 128 vertical bars and the bottom of every screenshot represent the state of the memory, with black representing 0 and lighter color corresponding to higher values of a given memory cell.

Each of this games offers a distinct challenge. Breakout is a relatively easy game with player's actions limited to moves along the horizontal axis. We picked Breakout because disastrous results of learning would indicate a fundamental problem with the RAM learning.

The deep Q-network for Seaquest constructed in [3] plays at an amateur human level and for this reason we consider this game as a tempting target for improvements. The game state has also some elements that possibly can be detected by the RAM-only network (e.g. oxygen-level meter or the number of picked divers).

Bowling seems to be a hard game for all deep Q-network models. It is an interesting target for the RAM-based networks, because visualizations suggest that only a couple of RAM cells are changing during play.

3.3. Technical infrastructure

For our experiments we adapted Nathan Sprague's implementation of deep Q-learning [48] in Theano [24] and Lasagne [49]. The code used for experiment uses Arcade Learning Environment [6] for communicating with an Atari simulator, but we also integrated our code with OpenAI gym, which is a wrapper on top of ALE. Our code can be found on github [50] (as well as in the directory `code/dqn` on the CD).

The experiments were performed on a GPU cluster of Interdisciplinary Centre for Mathematical and Computational Modelling using Linux machines equipped with NVIDIA GTX 480. Each of them lasted 1–3 days, with the screen-only models taking approximately twice as much time to train as RAM-only ones. The details of using the cluster's architecture can be found in appendix A.

3.4. Hyperparameters and network architectures

We have evaluated the performance of the DQN method for two neural network architectures. They serve as a Q-value approximators: accepting RAM as the input and returning estimates of Q-values of each possible action. We also made a limited number of experiments using networks accepting as an input both RAM and screen, which are described in section 4.4.

All the hyperparameters of the models we consider are the same as in [3] if not mentioned otherwise (see appendix B). One change we made is to decrease of the size of the replay memory (see section 2.4) to 10^5 observations, so that it fits 1.5GB of NVIDIA GTX 480 memory.[1]

We also decided to pass to the network only the RAM state corresponding to the last timestep before agent's action (and not 4, as in [3], compare section 2.3), as the reason to pass multiple frames as an input—violation of markovian property of environment—is void with RAM inputs. Basic experiments we performed suggest it was a good decision, as passing the last couple of RAM frames obstructed the learning procedure and worsened the results. We still used non-unity frameskip though: it is further discussed in section 4.3.

The RAM input to the network was scaled by 256, to make all the inputs lie between 0 and 1.

3.4.1. `just_ram`

The first proposed architecture is a small 2 hidden layer network with rectified linear units as an activation function.

Neural network 1: `just_ram` (numActions)

```
Input: RAM
Output: A vector of length numActions
hiddenLayer1 ← DenseLayer(RAM, 128, rectify)
hiddenLayer2 ← DenseLayer(hiddenLayer1, 128, rectify)
output ← DenseLayer(hiddenLayer2, numActions, no activation)
return output
```

3.4.2. `big_ram`

The second architecture consists of 4 hidden layers.

Neural network 2: `big_ram` (numActions)

```
Input: RAM
Output: A vector of length numActions
hiddenLayer1 ← DenseLayer(RAM, 128, rectify)
hiddenLayer2 ← DenseLayer(hiddenLayer1, 128, rectify)
hiddenLayer3 ← DenseLayer(hiddenLayer2, 128, rectify)
hiddenLayer4 ← DenseLayer(hiddenLayer3, 128, rectify)
output ← DenseLayer(hiddenLayer4, numActions, no activation)
return output
```

3.5. Evaluation method

The process of evaluating the RAM version of DQN is divided in experiments. By one experiment we mean a full training of a single deep Q-network in a fixed game. The training consists of 100 epochs, each containing a training and testing part.

During a training part of an epoch, agent plays the game for a couple of episodes (full games) for a total of 50000 steps, choosing actions based on his Q-function estimate in each step (using ε-greedy strategy). After a step, the learning algorithm performs one step of parameter optimization, using gradients of the loss function of the random 32 observations[2]

[1] We made experiments with replay memory size of $5 \cdot 10^5$ and have not observed a statistically significant change in results.

[2] This hyperparameter is mentioned as `BATCH_SIZE` in the code and the hyperparameter table.

(observations contain the RAM state before the action, the action itself, the reward achieved and the RAM state after the action) from the replay memory.

After a training part, we perform a testing part of an epoch. It consists of playing a trained agent for 10000 steps using $\varepsilon = 0.05$ without changing its parameters, only to evaluate its current performance. This number of steps usually covered around 10–20 game episodes.

The numerical result of an experiment we present thorough this work is the average cumulative (non-discounted) reward in episode during a best testing epoch. To make sure this is a reliable metric, we performed experiments in Breakout involving testing the network with best-epoch parameters on a couple different random seeds (a given game episode is initialized randomly, to make games slightly different each time they are played) for 100000 steps. The results of this evaluation were consistently lower by about 30% among all of the methods we tried, including `nips`.

3.6. Results of the basic method

The evaluation results during the course of training in all three games of the basic methods described above (`nips`, `just_ram`, `big_ram`) are presented in figures 3.2 to 3.4.

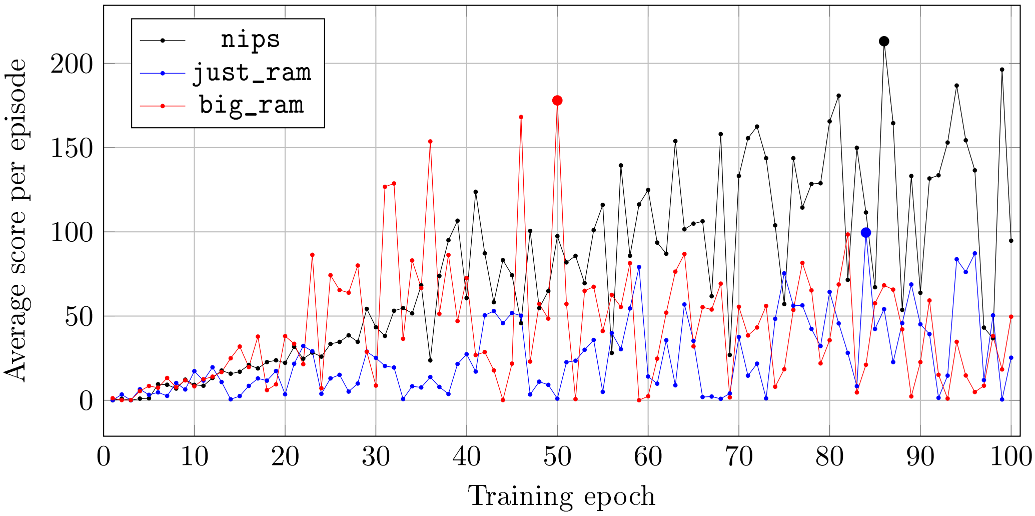


Figure 3.2: Training results for Breakout for three basic models: `nips`, `just_ram`, `big_ram`.

The best epoch results of these methods, along with `ale_ram`[3] are summarized in table 3.1.

	Breakout	Seaquest	Bowling
`nips` best	**213.14**	1808	54.0
`just_ram` best	99.56	1360	58.25
`big_ram` best	178.0	**2680**	**66.25**
`ale_ram`	4.0	593.7	29.3

Table 3.1: Table summarizing test results for basic methods.

[3] The `ale_ram`'s evaluation method differ—the scores presented are the average over 30 trials consisting of a long period of learning and then a long period of testing, nevertheless the results are much worse than of any DQN-based method presented here.

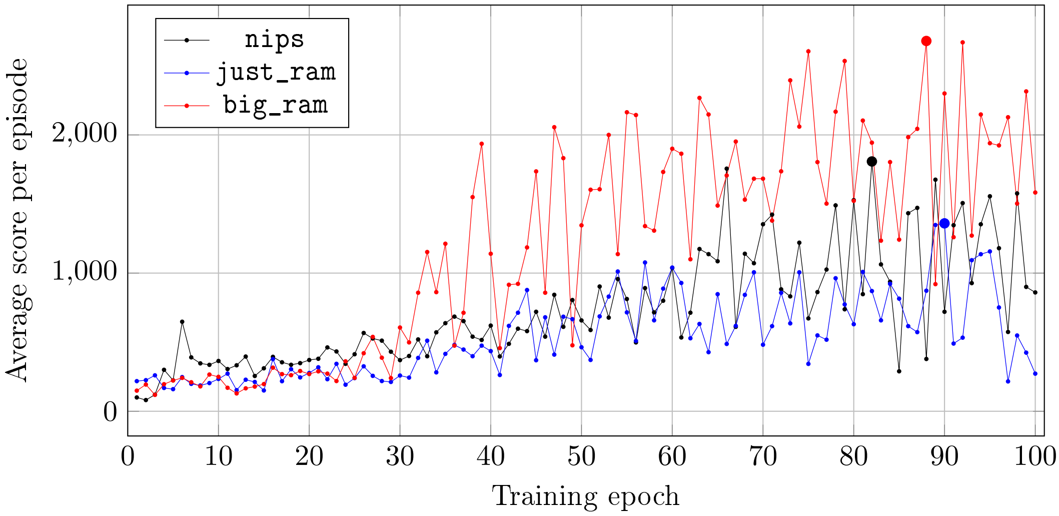


Figure 3.3: Training results for Seaquest three basic models: `nips`, `just_ram`, `big_ram`.

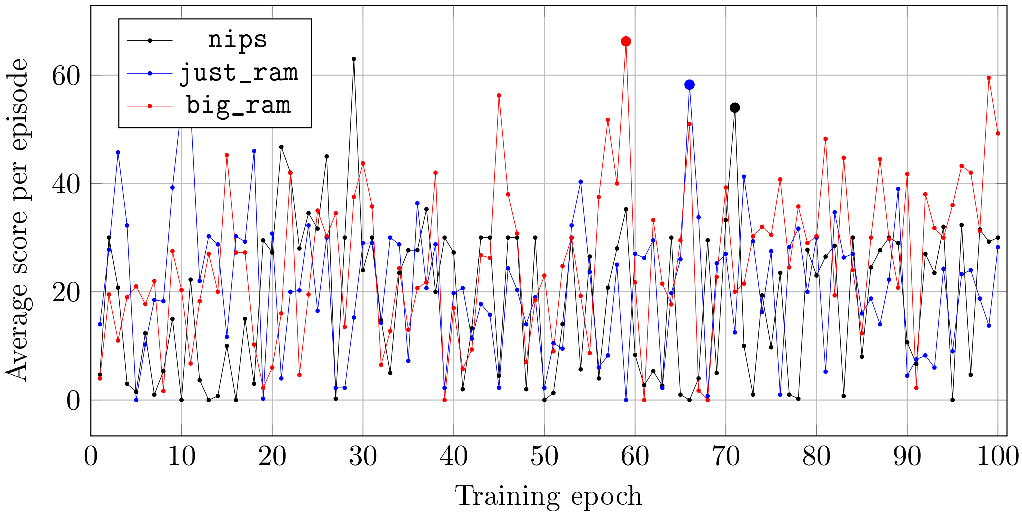


Figure 3.4: Training results for Bowling for three basic models: `nips`, `just_ram`, `big_ram`.

In Breakout, the best results for the `big_ram` model is weaker than one obtained by screen-only network `nips`. In Seaquest the `big_ram` model was better, and `just_ram` worse than `nips`, while in Bowling both RAM-based models performed better than screen-based `nips`. The training plots show that there is a big variance of the results between training epochs.

Chapter 4

Extensions

In the same spirit as [3] with experience replay and frameskip, we try a couple of extensions to the basic algorithm of DQN.

4.1. Dropout

Training a basic RAM-only network leads to high variance of the results (see the figures 3.2 to 3.4) over epochs. This can be a sign of overfitting. To tackle this problem we applied dropout [51], a standard regularization technique for neural networks.

Dropout is a simple, yet effective regularization method. It consists of "turning off" with probability p each neuron in training, i.e. setting the output of the neuron to 0, regardless of its input. In backpropagation, the parameters of switched off nodes are not updated. Then, during testing, all neurons are set to "on"—they work as in the course of normal forward pass, with the exception that each neuron's output is multiplied by p to make up for the skewed training.

The intuition behind the dropout method is that it forces each node to learn in absence of other nodes. The work [52] shows an experimental evidence that the dropout method indeed reduces the variance of the learning process.

We have enabled dropout with probability of turning off a neuron $p = \frac{1}{2}$. This applies to all nodes, except output ones. We implemented dropout for both RAM-only networks: `just_ram` and `big_ram`. The progress of training for Seaquest is show in figure 4.1 and the best epoch results are presented in the table 4.1. This method offers an improvement for the `big_ram` network for Seaquest, leading to the best results for this game in this paper.

	Breakout	Seaquest	Bowling
`just_ram with dropout` best	130.5	1246.67	54.75
`big_ram with dropout` best	122.25	**2805**	58.25

Table 4.1: Summary of test results for training which involves dropout with the parameter $p = 0.5$.

4.2. Learning rate

The high variance in training, as illustrated by figures figs. 3.2 to 3.4, could be caused by setting a too high learning rate. It may come from "stepping over" optimal values when

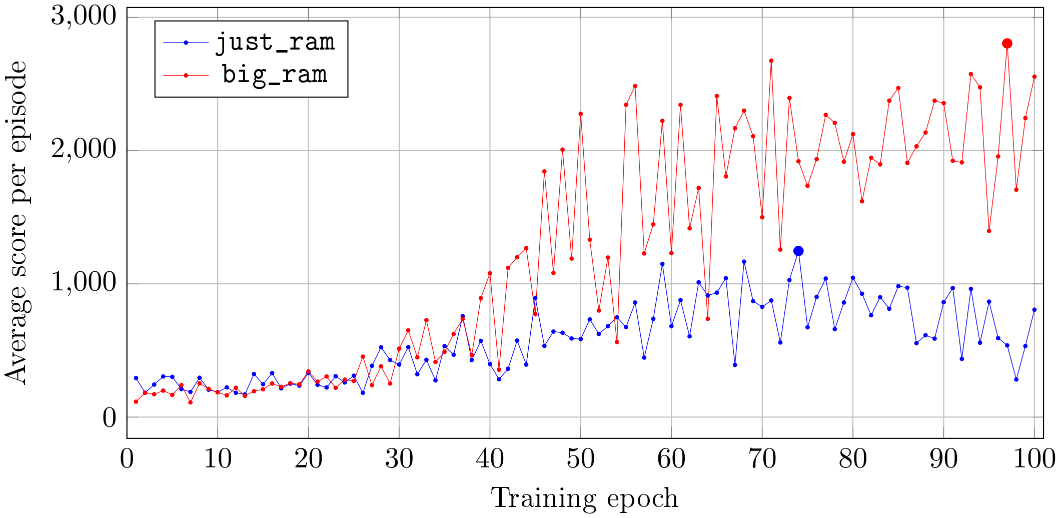


Figure 4.1: Training results for Seaquest with dropout $p = 0.5$ for models `just_ram`, `big_ram`. The figure suggests that indeed dropout reduces the variance of the learning process.

taking too big steps in the parameter space by the RMSProp algorithm. If it was the case, decreasing the step size should lead to slower learning combined with higher precision of finding minima of the loss function and better results.

We performed the experiments with reduction of learning rate from 0.0002 to 0.0001. The results of them can be found in table 4.2. Comparing to the training without dropout, scores improved only in the case of Breakout and the `just_ram` network, but not by a big margin.

	Breakout	Seaquest	Bowling
`just_ram` best	137.67	1233.33	51.5
`big_ram` best	112.14	2675	59.25

Table 4.2: Summary of test results for modified learning rate.

4.3. Frameskip

In the benchmark agent `nips`, authors use the frameskip (see section **??** for definition of frameskip) value of 4 and pass for training *all* 4 frames before the action to handle "blinking" behavior of Atari machine, i.e. showing some objects only every couple of frames. In the case of learning from memory there is no loss of information when previous RAM states are ignored. Hence in our models we only pass to the network the RAM state corresponding to the last frame before a given action. We made some limited experiments passing all previous `frameskip` RAM states to the network, but the results were worse than when passing only the last one. We still used frameskip of 4 in the basic experiments.

The work [53] suggests that choosing the right frameskip has a big influence on the performance of a model (see also [54]). We decided to try two bigger values of frameskip: 8 and 30, corresponding to deciding on actions 5 and 2 times a second.

The figure 4.2 and table 4.3 show a significant improvement of the `just_ram` model in Seaquest, as well as the `big_ram` model for Bowling. Quite surprisingly, the variance of the results appeared to be much lower with higher frameskip for all tested models.

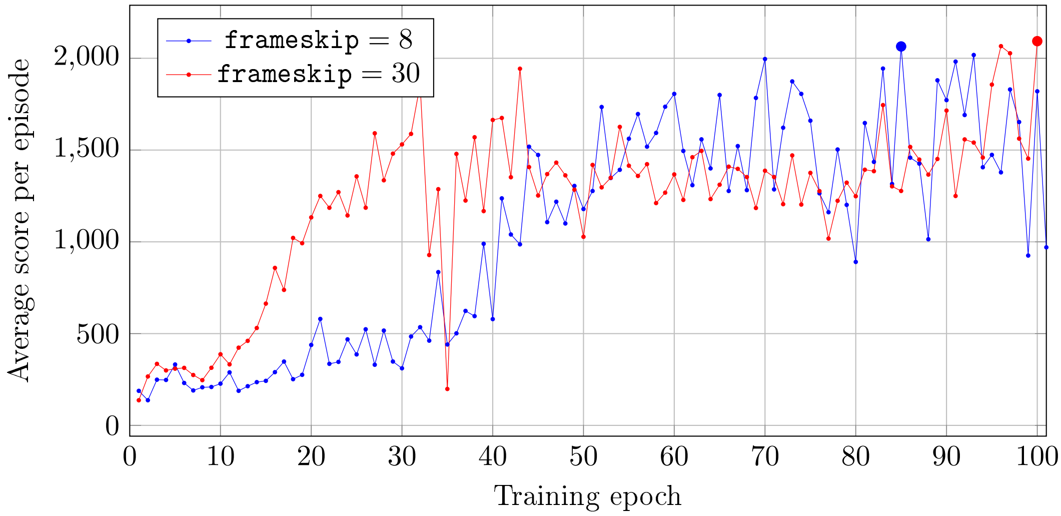


Figure 4.2: Training results for Seaquest with **frameskip** = 8 and **frameskip** = 30 for the model `just_ram`.

	Breakout	Seaquest	Bowling
`just_ram` with `frameskip` 8 best	82.87	2064.44	54.56
`just_ram` with `frameskip` 30 best	-	2093.24	54.36
`big_ram` with `frameskip` 8 best	102.64	2284.44	**70.78**
`big_ram` with `frameskip` 30 best	-	2043.68	61.9

Table 4.3: Table summarizing test results for training which involves higher `frameskip` value. For Breakout `frameskip` = 30 does not seem to be a suitable choice.

As noticed in [53], in the case of Breakout high frameskips, such as 30 lead to a disastrous performance, thus we tested only frameskip 8 and for it we received slightly weaker results than with a baseline frameskip 4.

4.4. Joining RAM and screen

An interesting question regarding learning agents playing Atari games is whether it is possible to combine information from the screen and RAM and create an agent that would successfully use both sources of information. This subsection makes first steps to towards answering this question.

We consider two mixed network architectures. The first one is `mixed_ram`, where we just concatenate output of the last hidden layer of the convolutional network with the RAM input and then apply a linear transformation with no non-linearity. It is worth to remind that the RAM input, as well as screen are scaled down to be in range $[0, 1]$; with no normalization a disastrous performance was observed.

Neural network 3: `mixed_ram` (numActions)

```
Input: RAM, screen
Output: A vector of length numActions
conv1 ← Conv2DLayer(screen, rectify)
conv2 ← Conv2DLayer(conv1, rectify)
hidden ← DenseLayer(conv2, 256, rectify)
concat ← ConcatLayer(hidden, RAM)
output ← DenseLayer(concat, numActions, no activation)
return output
```

The other architecture is a deeper version of `mixed_ram`. We use more dense layers which are applied in a more sophisticated way as described below.

Neural network 4: `big_mixed_ram` (numActions)

```
Input: RAM, screen
Output: A vector of length numActions
conv1 ← Conv2DLayer(screen, rectify)
conv2 ← Conv2DLayer(conv1, rectify)
hidden1 ← DenseLayer(conv2, 256, rectify)
hidden2 ← DenseLayer(RAM, 128, rectify)
hidden3 ← DenseLayer(hidden2, 128, rectify)
concat ← ConcatLayer(hidden1, hidden3)
hidden4 ← DenseLayer(concat, 256, rectify)
output ← DenseLayer(hidden4, numActions, no activation)
return output
```

The results, which are presented in table 4.4 are reasonable, but not impressive. In particular, we did not see any improvement over the benchmark `nips` agent, which is embedded into both mixed architectures. This suggests that in `mixed_ram` and `big_mixed_ram` models the additional information about the RAM state is not used in a productive way.

	Breakout	Seaquest	Bowling
`mixed_ram` best	143.67	488.57	39.5
`big_mixed_ram` best	67.56	1700	45.5

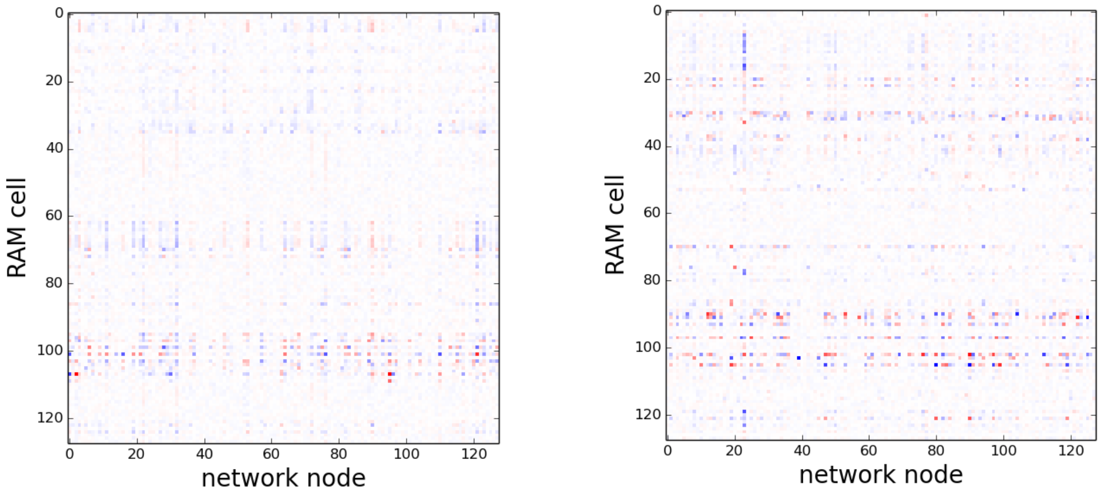
Table 4.4: Table summarizing test results for methods combining information from the screen and from the memory.

4.5. RAM visualization

To better understand how the agents work, we visualized in figure 4.3 the first layers of the network `big_ram` in Breakout and Seaquest.

Each column corresponds to one of 128 nodes in the first layer of the trained network and each row corresponds to one of 128 memory cells. The color of a grid cell describes whether the high value in a given RAM cell negatively (blue) or positively (red) influences the activation level for a given neuron.

Figure 4.3 suggests that the RAM cells with numbers 95–105 are important for gameplay—the activation of many neurons of the `big_ram` network depend on the state of these cells.



(a) `big_ram` in Breakout

(b) `big_ram` in Seaquest

Figure 4.3: Visualization of the parameters of the first layer of the trained Q-networks.

Chapter 5

Conclusions and future work

We trained two neural network agents as well as their variations to play Atari 2600 games: Breakout, Seaquest and Bowling. The agents used not the screen, but the information stored in the memory of the console to decide on moves. In all games, the RAM agents were on a par with the screen-only agent `nips` presented in [3].

In the case of Seaquest and Bowling even a simple `just_ram` model with an appropriately chosen frameskip parameter, as well as a bigger model `big_ram` performed better than the benchmark agent. In the case of Breakout, the results are below the `nips` agent, but still were reasonable.

5.1. Future work

Due to limitations in the computing power we had, as well as the novel aspect of the research, our study has a preliminary character. Here we present a number of ideas to extend our work.

5.1.1. Other games

The number of Atari games can be counted in hundreds, yet we only tested our agents on 3 of them. It may be interesting to see how the results transfer to other games, like Pacman or Space Invaders and what aspects of the game tactics the agents are able to learn in these games.

5.1.2. Better architectures and hyperparameters

The works [4, 39, 40, 41] introduced more involved ideas to improve deep Q-learning. We would like to see if these improvements transfer to the RAM models.

It would be also interesting to spend more time tuning hyperparameters to possibly uncover the real potential of our method. In particular, we are interested in:

- trying more complex, deeper architectures for processing RAM,
- improving stability of learning and reducing the variance, potentially with the usage of batch normalization [55],
- more effective joining of information from the screen and memory,
- considering unsupervised pretraining, using practically infinite stream of RAM states to learn an autoencoder, that can help to better initialize our models [56].

5.1.3. Recurrent neural network

To train agents in non-markovian environments (called *partially observable* Markov decision processes), one can use a recurrent neural network that is able to "remember" some function of the previous states [57].

While RAM-based models satisfy markovian assumption (as argued in section 4.3), it may still be beneficial to the progress of training to model Q-function using a RNN. This way, agent could e.g. decide that it will be repeating a given action or following a pattern of actions without the need to constantly receive encouraging signal from the RAM state.

Appendix A

ICM cluster setup

Most of the experiments we executed during the project were ran on a cluster of Interdisciplinary Centre for Mathematical and Computational Modelling of University of Warsaw (known as ICM). We got a lot of experience using these machines and decided to write it up in this appendix in the hope that it may be a useful guide for someone else doing a deep learning project using the ICM's architecture.

A.1. Hardware architecture

ICM owns a couple of computing clusters for different purposes. We used the only one containing GPU cards: Grom [58, in polish]. It contains an ssh server and a number of computational nodes. You can ssh to the ssh server only, and from there you can insert a job into a Slurm [59] queue. A job can be an interactive bash session, which is very convenient to try things out.

The specification of the job uses Slurm format, but the commands to execute can be any bash commands. One flag worth to mention is `--gres`, which allows to specify request for GPU. You can see an example Slurm job specification in listing A.1.

```
#!/bin/bash -l
#SBATCH -J dqn
#SBATCH -N 1
#SBATCH --ntasks-per-node 8
#SBATCH --mem 4000
#SBATCH --time=165:00:00
#SBATCH --gres=gpu:1
#SBATCH --output="logs"
#SBATCH --exclude wn8002,wn8006
#SBATCH -C cuda

source /icm/home/sygnowsk/dqn/deep_q_rl/venv/bin/activate
python run_sygi.py -r enduro.bin --network-type big_ram
```

Listing A.1: Example Slurm job specification

The job will be executed on one[1] of the 24 computational nodes, each having 2 NVIDIA GTX 480 cards. During our experiments, to fully utilize the resources, we limited

[1] It is possible to reserve more nodes for one job, but we didn't use that feature.

the memory and GPU number requirements to be able to run two experiments in each node in parallel.

A.2. Software stack

The operating system installed on the Grom cluster is CentOS6. The main system is quite raw, and to make use of it, one needs to use modules (loaded using command `module load NAME`) with compilers, libraries, etc.

During the first try to run DQN we wanted to use the code provided by the authors of the algorithm, which was written using Torch. Even though Lua, as well as its package manager luarocks was installed in the system, we were having hard time installing the code dependencies without root privileges.

Due to these problems, we decided to move to code based on Theano, which is based on python. We downloaded a standalone python script enabling `virtualenv`, which allows to encapsulate a project's python dependencies in a local directory, making it possible to install python libraries locally. The only library we had to install from sources was OpenCV, whose python version: `opencv2` is just a wrapper around the regular OpenCV and expects the library to be already installed in the system.

A.3. Running experiments

The result of running the experiment was a directory with a file listing the results of training after each epoch and one file per epoch with the network parameters, as well as a Slurm file with output from stdout and stderr. The parameters were serialized using python package `pickle`.

As one job usually took a day or two to finish, it was often the case that there was a number of jobs in progress and it was hard to distinguish, which was which. To tackle this problem we updated the code to output not only evaluation results, but also all the hyperparameters of the model.

After we finished our experiments, a company deepsense.io released a tool: Neptune, which aims to simplify management of experiments.

Appendix B

Hyperparameters

The list of hyperparameters and their descriptions. Most of the descriptions come from [4].

hyperparameter	value	description
minibatch size	32	Number of training cases over which each stochastic gradient descent (SGD) update is computed.
replay memory size	100 000	SGD updates are randomly sampled from this number of most recent frames.
phi length	4	The number of most recent frames experienced by the agent that are given as input to the Q network in case of the networks that accept screen as input.
update rule	`rmsprop`	Name of the algorithm optimizing the neural network's objective function
learning rate	0.0002	The learning rate for rmsprop
discount	0.95	Discount factor γ used in the Q-learning update. Measures how much less do we value our expectation of the value of the state in comparison to observed reward.
epsilon start	1.0	The probability (ε) of choosing a random action at the beginning of the training.
epsilon decay	1000000	Number of frames over which the ε is faded out to its final value.
epsilon min	0.1	The final value of ε, the probability of choosing a random action.
replay start size	100	The number of frames the learner does just the random actions to populate the replay memory.

Table B.1: Hyperparameters used by our models.

Bibliography

[1] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016). Article, pp. 484–489. ISSN: 0028-0836. URL: http://dx.doi.org/10.1038/nature16961.

[2] Jakub Pawlewicz Henryk Michalewski Andrzej Nagórko. "485 - A New Upper Bound for Morpion Solitaire." In: CGW/GIGA@IJCAI (2015).

[3] Volodymyr Mnih et al. "Playing Atari With Deep Reinforcement Learning". In: *NIPS Deep Learning Workshop*. 2013.

[4] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. URL: http://dx.doi.org/10.1038/nature14236.

[5] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *arXiv preprint arXiv:1602.01783* (2016).

[6] M. G. Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279.

[7] Jakub Sygnowski and Henryk Michalewski. "Learning from the memory of Atari 2600". In: *arXiv preprint arXiv:1605.01335* (2016).

[8] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 1998.

[9] Christopher J. C. H. Watkins and Peter Dayan. "Technical Note Q-Learning". In: *Machine Learning* 8 (1992), pp. 279–292. DOI: 10.1007/BF00992698. URL: http://dx.doi.org/10.1007/BF00992698.

[10] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". In: *Machine Learning* 8.3 (), pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698. URL: http://dx.doi.org/10.1007/BF00992698.

[11] John N Tsitsiklis. "Asynchronous stochastic approximation and Q-learning". In: *Machine Learning* 16.3 (1994), pp. 185–202.

[12] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[13] Frank Rosenblatt. *Principles of neurodynamics; perceptrons and the theory of brain mechanisms.* 616 p. Washington: Spartan Books, 1962, 616 p. URL: //catalog.hathitrust.org/Record/000203591.

[14] Gao Huang et al. "Deep Networks with Stochastic Depth". In: *CoRR* abs/1603.09382 (2016). URL: http://arxiv.org/abs/1603.09382.

[15] Andrej Karpathy. *CS231n. Neural Networks Part 1: Setting up the Architecture.* URL: http://cs231n.github.io/neural-networks-1/#actfun.

[16] Augustin Cauchy. "Méthode générale pour la résolution des systèmes d'équations simultanées". In: *Compte rendu des séances de l'académie des sciences* (1847), pp. 536 –538. URL: https://cs.uwaterloo.ca/~y328yu/classics/cauchy-en.pdf.

[17] *The dataset of the COCO challenge.* URL: http://mscoco.org/dataset/#download.

[18] *Nvidia CUDA documentation.* URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide.

[19] Ning Qian. "On the momentum term in gradient descent learning algorithms". In: *Neural Networks* 12.1 (1999), pp. 145 –151. ISSN: 0893-6080. DOI: http://dx.doi.org/10.1016/S0893-6080(98)00116-6. URL: http://www.sciencedirect.com/science/article/pii/S0893608098001166.

[20] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *Proceedings of the 3rd International Conference on Learning Representations (ICLR).* 2014.

[21] Matthew D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method". In: *CoRR* abs/1212.5701 (2012). URL: http://arxiv.org/abs/1212.5701.

[22] Geoffrey Hinton. *Neural networks for machine learning, lecture 6e.* URL: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. "Deep Learning". Book in preparation for MIT Press. 2016. URL: http://www.deeplearningbook.org.

[24] Theano Development Team. "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv e-prints* abs/1605.02688 (May 2016). URL: http://arxiv.org/abs/1605.02688.

[25] *Contribution to Theano of user sygi.* URL: https://github.com/Theano/Theano/pulls?q=is%3Apr%20author%3Asygi%20.

[26] R. Collobert, K. Kavukcuoglu, and C. Farabet. "Torch7: A Matlab-like Environment for Machine Learning". In: *BigLearn, NIPS Workshop.* 2011.

[27] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[28] François Chollet. *keras.* 2015. URL: https://github.com/fchollet/keras.

[29] Bart van Merriënboer et al. "Blocks and Fuel: Frameworks for deep learning". In: *CoRR* abs/1506.00619 (2015). URL: http://arxiv.org/abs/1506.00619.

[30] Aymeric Damien et al. *TFLearn.* 2016. URL: https://github.com/tflearn/tflearn.

[31] Wikipedia. *Atari 2600 — Wikipedia, The Free Encyclopedia.* [Online; accessed 20-October-2016]. 2016. URL: https://en.wikipedia.org/w/index.php?title=Atari_2600&oldid=745317391.

[32] Chriskohler. *Racing the Beam: How Atari 2600's Crazy Hardware Changed Game Design.* URL: https://www.wired.com/2009/03/racing-the-beam.

[33] *Stella: A multi-platform Atari 2600 VCS emulator.* URL: http://stella.sourceforge.net/.

[34] Greg Brockman et al. *OpenAI Gym.* 2016. eprint: arXiv:1606.01540.

[35] Tejas D. Kulkarni et al. "Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation". In: *CoRR* abs/1604.06057 (2016). URL: http://arxiv.org/abs/1604.06057.

[36] Sven Koenig and Reid G. Simmons. "The Effect of Representation and Knowledge on Goal-Directed Exploration with Reinforcement-Learning Algorithms". In: *Machine Learning* 22.1 (1996), pp. 227–250. ISSN: 1573-0565. DOI: 10.1023/A:1018068507504. URL: http://dx.doi.org/10.1023/A:1018068507504.

[37] *Boxing (1980 video game)*. URL: https://en.wikipedia.org/wiki/Boxing_(1980_video_game) (visited on 10/28/2016).

[38] Nir Lipovetzky, Miquel Ramirez, and Hector Geffner. "Classical Planning with Simulators: Results on the Atari Video Games". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2015.

[39] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double Q-learning". In: *arXiv preprint arXiv:1509.06461* (2015).

[40] Yitao Liang et al. "State of the Art Control of Atari Games Using Shallow Reinforcement Learning". In: *arXiv preprint arXiv:1512.01563* (2015).

[41] Ziyu Wang et al. "Dueling Network Architectures for Deep Reinforcement Learning". In: *arXiv preprint arXiv:1511.06581* (2015).

[42] *Bowling (video game)*. URL: https://en.wikipedia.org/wiki/Bowling_(video_game) (visited on 04/15/2016).

[43] *The Bowling manual*. URL: https://atariage.com/manual_html_page.php?SoftwareID=879 (visited on 04/25/2016).

[44] *Breakout (video game)*. URL: https://en.wikipedia.org/wiki/Breakout_(video_game) (visited on 04/15/2016).

[45] *The Breakout manual*. URL: https://atariage.com/manual_html_page.php?SoftwareID=889 (visited on 04/25/2016).

[46] *Seaquest (video game)*. URL: https://en.wikipedia.org/wiki/Seaquest_(video_game) (visited on 04/15/2016).

[47] *The Seaquest manual*. URL: https://atariage.com/manual_html_page.html?SoftwareLabelID=424 (visited on 04/25/2016).

[48] *Nathan Sprague's implementation of DQN*. URL: https://github.com/spragunr/deep_q_rl (visited on 04/15/2016).

[49] *Lasagne - lightweight library to build and train neural networks in Theano*. URL: https://github.com/lasagne/lasagne.

[50] *The repository of our code*. URL: https://github.com/sygi/deep_q_rl.

[51] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[52] David Warde-Farley et al. "An empirical analysis of dropout in piecewise linear networks". In: ICLR2014. 2014. URL: http://arxiv.org/abs/1312.6197.

[53] Alex Braylan et al. "Frame Skip Is a Powerful Parameter for Learning to Play Atari". In: *AAAI-15 Workshop on Learning for General Competency in Video Games*. 2015. URL: http://nn.cs.utexas.edu/pub-view.php?PubID=127530.

[54] Aaron Defazio and Thore Graepel. “A Comparison of learning algorithms on the Arcade Learning Environment”. In: *CoRR* abs/1410.8620 (2014). URL: http://arxiv.org/abs/1410.8620.

[55] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *CoRR* abs/1502.03167 (2015). URL: http://arxiv.org/abs/1502.03167.

[56] Pascal Vincent et al. “Extracting and Composing Robust Features with Denoising Autoencoders”. In: *Proceedings of the 25th International Conference on Machine Learning*. ICML ’08. New York, NY, USA: ACM, 2008, pp. 1096–1103. ISBN: 978-1-60558-205-4. DOI: 10.1145/1390156.1390294. URL: http://doi.acm.org/10.1145/1390156.1390294.

[57] Junhyuk Oh et al. “Control of Memory, Active Perception, and Action in Minecraft”. In: *CoRR* abs/1605.09128 (2016). URL: http://arxiv.org/abs/1605.09128.

[58] *Architektura systemu Grom*. URL: https://www.icm.edu.pl/kdm/Grom.

[59] *Slurm: workload manager*. URL: http://slurm.schedmd.com/.