University of Tartu
Faculty of Mathematics and Computer Science
Institute of Computer Science

Kristjan Korjus, Ilya Kuzovkin, Ardi Tampuu, Taivo Pungas

Replicating the Paper "Playing Atari with Deep Reinforcement Learning" [MKS+13]

Technical Report

MTAT.03.291 Introduction to Computational Neuroscience

Tartu 2014

Contents

Introduction

In the recent years the popularity of the method called *deep learning*[Hin07] has increased noticeably in the machine learning community. Deep leaning was successfully applied to speech recognition[DYDA12] and many other tasks in machine learning[DHK13]. In all these studies the performance of the resulting system was better than other machine learning methods were able to achieve so far.

The core of the deep learning method is an artificial neural network. One of the properties, which gives this family of learning algorithms a special place, is the ability to extract a "meaningful" (from the human perspective) *concepts* from the data by combining the features based on the structure of the data. The extracted concepts sometimes have clear interpretation and that makes us feel as if the machine has indeed *learned* something. Here we step into the realm of artificial intelligence, the possibility of which never stops to fascinate our minds.

One of the recent works, which brings together deep learning and artificial intelligence is a paper "Playing Atari with Deep Reinforcement Learning"[MKS+13] published by DeepMind[1] company. The paper describes a system, which combines deep learning methods and the *reinforcement learning* in order to create a system that is able to learn how to play simple computer games. It is worth mentioning that the system has access only to the visual information (screen of the game) and the scores. Based on these two inputs the system learns to understand which moves are good and which are bad depending on the situation on the screen. Notice that the human player uses exactly same information to evaluate his performance and adapt his playing strategy. The reported result shows that the system was able to master several different games and play some of them better than the human player.

This result can be seen as a step forward to the truly intelligent machines and thus it fascinates us. The goal of this project is to create an open-source analogue of such a system using the description provided in the paper.

[1] `http://deepmind.com`

1 Overview of the system

Before we go into the details, let us describe the overall architecture of the system and show how the building blocks are put together.

1.1 The task

The system receives a picture of a game screen (an example is shown in Figure 1) and chooses an action to take. It then executes this action and is told whether the score increased, decreased or did not change. Based on this information and playing a large number of games, the system needs to learn to improve its performance in the game.

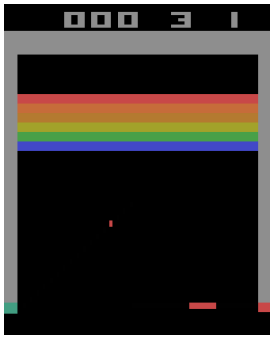

Figure 1: A game screen of Breakout.

1.2 Reinforcement learning

In a reinforcement learning setting, an agent takes actions in an environment with the goal of maximising a cumulative reward. We tried to create a software agent that plays ATARI games in an emulator (the environment) and maximises its performance in the game, measured by its score in the games.

1.2.1 Exploration-exploitation

When the algorithm chooses between possible actions, it picks a "learned" action with probability $1 - \epsilon$ and a random action with probability ϵ. The value of ϵ is gradually decreased as the algorithm learns to play better.

It is necessary to sometimes pick a random action to not get stuck in local reward maxima. At first, the value of ϵ is small and the agent takes random actions most of the time. This relatively high emphasis on exploration is necessary for the agent to collect information about the environment. When ϵ starts to decrease, the agent starts to apply its learned behaviour more and more, i.e. moving towards exploitation.

ϵ never reaches zero in the case of our agent, so it still does some exploration even when performing well in the game.

1.3 Neural network

The system uses a neural network to assign an expected reward value to each possible action. The input to the network at any time point consists of the last four preprocessed game screens the system received. This input is then passed through three successive hidden layers to the output layer.

The output layer has one node for each possible action and the activation of those nodes indicates the expected reward from each of the possible actions - here, the action with the highest expected reward is selected for execution.

1.4 Learning process

Under the fancy word "neural network" hides a quite simple idea: bunch of *nodes* (neurons), each of them having some *input* to perform a computation on and an *output* where the result of the computation will be sent to, are connected to each other. Each connection has a *weight*, which regulates how much one neuron can affect another. The very first layer of the network is usually called the *input layer*. This is the place where we inject a data sample. Next to the input layer the network has several trickily connected *hidden layers*. And the last layer is an *output layer*, it gives us the final piece of information we wanted to know about the data sample we fed into the network. The resulting structure can be very complex, but the building blocks are always the same: neurons and connections between them.

We, as the builders of the system, usually know what we want it do: that is for each data sample we know what the final output should look like. Now the "smart" neural network is the one able to produce the output we expect, the "naive" network is the one which is not. The only thing that differs in those two are the *weights* on the connections between the nodes. By changing them in accordance with our final goal the network goes from being "naive" to being "smart". This process in general in known as *learning*.

To explain the concept of learning to the machine we introduce the notion of *cost function* (or *loss function*) which we will denote by L. Given the parameters of the network this function goes over the data samples, computes the outputs, compares these outputs with the expected ones and calculates how big is the error the system makes. The learning process can be represented as the process of *minimizing* the cost function.

The obvious way to minimize a function is to try out all possible inputs to find the minimal output. Unfortunately the number of possible input is unfeasibly large. How to do it is a big question not only for our system. The whole brach on computer science called *optimization* is dealing with this issue. One of the most popular technique used in this area is called *gradient descent*.

The idea of gradient descent is rather trivial: at each *iteration* of the learning algorithm it makes a small step in the "direction" which makes the value of the loss function smaller. Thus by making enough steps the algorithm will reach an *optimum*: a place from where step in any direction will only increase the value of the loss function. Once this optimum is found we say that the learning has finished and the system is now as smart as it can be. It might happen that the optimum we found is not the best one (*local optimum*), and there are bunch of techniques (for example *Monte Carlo* methods) to find the *global*

optimum, but we will not dwell into this right now.

Gradient descent uses the *gradient* (multidimensional derivative) as it's core:

1. Let the current configuration of the system be $\mathbf{w_0}$
2. Compute the derivative (gradient) of the loss function at this point $L(\mathbf{w_0})$
3. Look at the derivative with respect to each element $(w_1, \ldots, w_n) = \mathbf{w_0}$ and:
   (a) If the derivative is 0 then we should not touch this parameter of our system
   (b) Otherwise move one step in the direction opposite to the derivative (positive slope – decrease the parameter, negative slope – increase the parameter)
4. Repeat starting from the step 2. until the derivatives with respect to all parameters of the system are zeros (or close to zeros)

Once this process is complete we believe that our system has reached the optimal state: it's parameters are configures in such way that it will give optimal performance on the dataset we have. In our system we will use variation of the idea of gradient descent called RMSProp, you will read about it later.

2 Components of the system

In this section we indent to describe all the key components of the system in the detail. Description should reflect all the choices we made, justify them and explain how the component works.

2.1 Launching and communicating with ALE

Just as we, humans, exchange information with a computer game by seeing the computer screen (input) and pressing the keys (output actions), the system needs to communicate with the Arcade Learning Environment that hosts the game.

The communication with ALE is achieved using two first-in-first out pipes which must be created before the ALE is launched. We create the pipes and launch ALE using the os package of Python. The parameters given to ALE at execution must specify the way we want to communicate with ALE (FIFO pipes) and the location of the binary file containing the game to run (Breakout). In addition, we specify that we want ALE to return unencoded images, that only every 4th frame should be sent to us and whether the game window should be made visible.

The actual communication starts with a handshake phase, where we define the desired inputs (screen image and episode information) and ALE responds by informing us about the dimensions of the image. Thereafter the conversation between ALE and our agent consists in reading and deciphering inputs from FIFO_in pipe and sending the chosen actions back through FIFO_out pipe. The information is read from pipes as one long String, so deciphering is needed (cutting the input and converting the pieces into appropriate types). Similarly, the chosen action has to be transformed to an output string of specific format. If a game is lost, a specific "reset" signal is sent to start the next game. When the desired number of games has been played, the communication with ALE can be terminated by closing the communication pipes.

2.2 Convolutional neural network

Our system receives 4x84x84 pixel values as input. In order to find relevant information from these four 84x84 images, we use convolutional neural networks (CNN) [reference???]. CNNs are a specific type of neural network that are particularly adapted to extract features from images.

Unlike RBMs which see their input as an 1D vector, a CNN treats the input images as 2D objects. These 2D matrices of pixel values are convolved with linear filters to obtain the activities of the next layer. For example, our first hidden layer is generated by convolving our images with sixteen different 8x8 filters, using a step of 4 pixels. This means that for each of the 16 different filters, we first take the topleft 8x8 values of an image and linearly combine them with the filter, obtaining as a result one activity value.

We then move 4 pixels left and multiply another 8x8 area from the image with the same filter (same weight values). When reaching the right edge of the image, we start again from the left, only 4 pixels lower. Each linear combination yields one activity value, so convolving an 84x84 image with an 8x8 filter with step 4 produces 20x20 topologically arranged values. As we apply 16 different filters, we end up with 16*20*20 nodes in the first hidden layer of our convolutional network.

An attentive reader notices that our input consists of 4 images, not only one. This means that each of our filters contains four weight matrices of size 8x8. The images will be convolved with the corresponding 8x8 matrix and the calculated activities are summed up, so we end up with 16*20*20 values as before. All in all, W_{ij}^{KL} denotes the weight value connecting the pixel value on coordinates (i,j) of the selected 8x8 area of the L-th image with the K-th feature map.

Last but not least, a linear rectifier is applied to the activity values, setting all negative values to 0.

The obtained sixteen 20x20 feature maps are thereafter convolved with 32 4x4 filters. Using a step of 2, this yields 32x9x9 activity values in the second hidden layer. As before, the filters are not 2-dimensional, instead they have 16x4x4 values.

With these two convolutional layers we have reduced the number of nodes from 4x84x84 to 32x9x9. Functionally, after training the system we expect the nodes of the second hidden layer to represent spatiotemporal patterns relevant for successfully playing the game.

2.3 Q-learning

...

2.4 Root Mean Squares of gradients (RMSProp)

In Section 1.4 we have seen the logic of gradient descent update procedure. There are, however, problems with this approach. One of them is that once you have established and update vector $\mathbf{u}$ you might notice that for some components $u_i = \frac{\partial L(\mathbf{w})}{\partial w_i}$ it proposes huge changes and tiny changes for the others. It might be not very reasonable to use the same learning μ rate for all components. The idea of RMSProp is to come up with a separate learning rate μ_i for each of the components u_i.

If you can use all of your data to estimate the direction of the gradient, then you can cope with this problem by just using sign of the gradient $sign(\frac{\partial L(\mathbf{w})}{\partial w_i})$. This will ensure that all components of the gradient are treated equally. This approach is called *rprop*. However we do not have the luxury of using all the data points in our loss function L: it

would take too much time. This is where the idea of *stochastic gradient descent* comes into play.

2.4.1 Stochastic gradient descent

As you know, if you have a set of differentiable functions, then the sum of these functions is also differentiable and the derivative of this function is equal to the sum of the derivatives of these functions:

$$\nabla f(\mathbf{w}, \mathbf{x}) = \sum \nabla g(\mathbf{w}, \mathbf{x_k})$$

where $\mathbf{x}$ is a set of data samples and $\mathbf{x_k}$ is one data sample. This fact allows us to take a sample from the dataset, compute updates $\mathbf{u}$ on it and then repeat this several time. In the end of the day it will produce the same results as if we had performed the gradient descent on the whole dataset at once.

Unfortunately rprop does not work with small sets of subsamples[TH12] (*minibatches*). Although such blunt simplification as taking a sign instead of the value work on whole dataset (because it effectively converges to the average value of the magnitude of the update) it does not work on small random subsamples: there we could easily obtain an update which is too large compared to what we would like to have. The next build-up on top of the rprop idea is called *rmsprop* and deals with that issue.

2.4.2 RMSProp

When we average the results obtained using rprop method we divide by the magnitude of the gradient. RMSProps proposes to keep track of previous gradients and divide updates not by the current magnitude, but the average magnitude over the last several updates (minibatches). This will allow to modify each component u_i according to its previous magnitudes, preventing from taking it into account with too large or too small weight.

3 Implementation details

The prototype of the system is built using `Python 2.7` with heavy usage of `theano`[BBB+10] and `numpy`[Oli07] libraries. Once the proof-of-concept stage is complete we will consider using `C++` if it will yield better performance. On Figure 2 we present current overall structure of the application. The subsections below will explain the particularities.

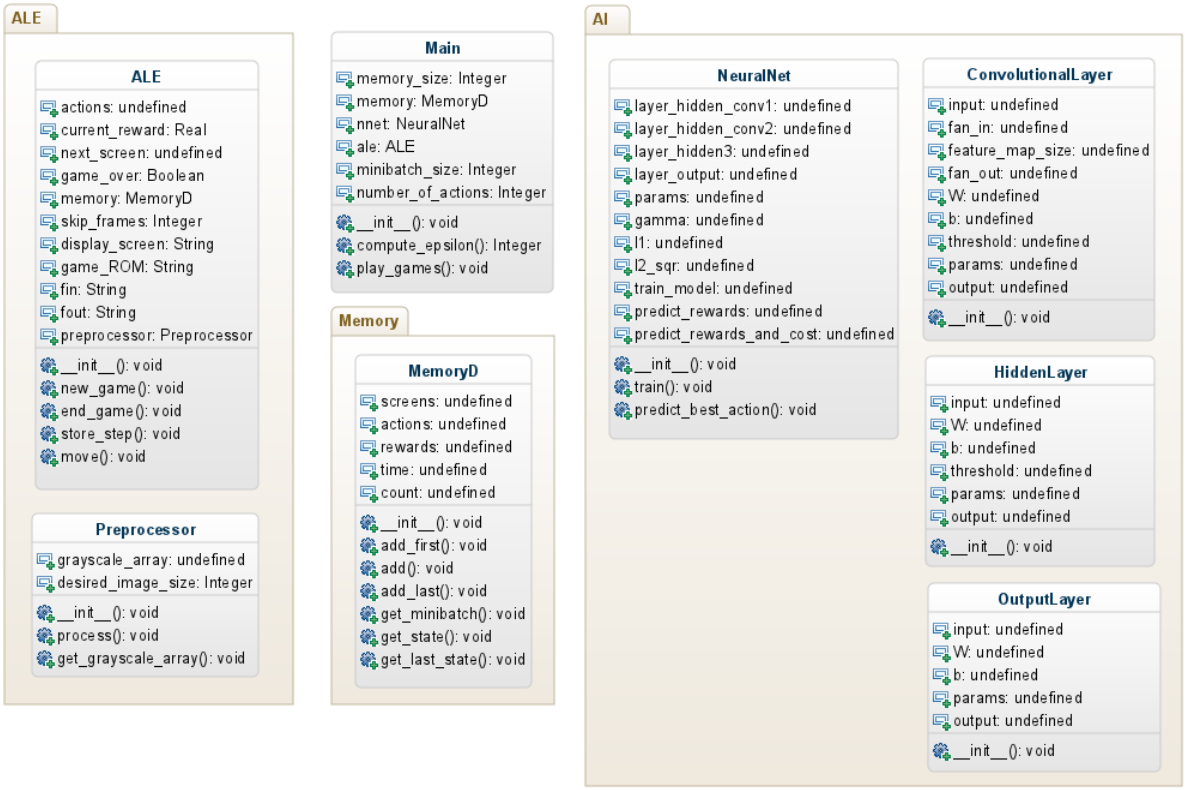

Figure 2: Class diagram representing the overall structure of the application.

3.1 Atari Learning Environment

To simulate the game we use Atari Learning Environment (ALE)[BNVB13]. It is a game simulator which allows to programmatically send player commands and receive the game output (image of the game screen, score, state of the game). Class `ALE` facilitates communication with the game emulator.

3.2 Preprocessing

The game screens are preprocessed by cropping the original 160x210-pixel image to a 160x160 region of interest, which is then downscaled to a 84x84 image.

The colors from ATARI's NTSC palette are converted to RGB using a conversion table [2]. The RGB representation is then converted to grayscale according to the weighted

[2] `http://www.biglist.com/lists/stella/archives/200109/msg00285.html`

combination $0.21R + 0.71G + 0.07B$. This should produce a representation close to human perception (humans are more sensitive to green than other colours)[3]. An example of a preprocessed image is shown in Figure 3.

Figure 3: A preprocessed game screen of Breakout.

3.3 Memory

On of the central parts of the applications if the memory, where game states, player actions and received rewards are stored. Class `MemoryD` is a structure where all this information is stored and implements methods for storing and extracting information.

3.4 Neural network

In our implementation of the neural network we heavily rely on the `Theano` toolbox. Theano allows to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently[4]. We closely follow the deep learning tutorial[5] to define and build the neural network. Classes `ConvolutionalLayer`, `HiddenLayer` and `OutputLayer` describe building block of the network and the `NeuralNet` class puts them together and provides function for training and using the neural network.

3.5 Computing on GPU

As you can imagine, the computations on a neural network are highly parallelizable due to the fact that you can compute each neuron independently. Also the learning process can be parallelized to evaluate several inputs at the same time to obtain several updates in one unit of time. Theano library is able to perform the highly parallel GPU computations.

3.6 Running instructions

For more technical details and running instruction please refer to our wiki `https://github.com/kristjankorjus/Replicating-DeepMind/wiki`

[3] `http://www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grayscale/`
[4] http://deeplearning.net/software/theano
[5] http://www.deeplearning.net/tutorial

4 Results

At this point of time we can not drive any conclusions about how well the system perform or is it functional at all. Before we will be able to estimate that we need to finish the system build-up and confirm that all the learning procedures work.

4.1 Performance measures

One possible measure will be to plot the score change over the time. If we will observe that over the time the system is able to score more during fixed-length time window than before, we will have a strong indication that the system is doing something reasonable.

4.2 Comparison to human player

Yet to be performed.

4.3 Comparison to the original paper

Yet to be performed.

4.4 Applications and future usage

We believe that the technique used in this system allows to solve wide variety of machine learning tasks. It will be interesting to test the system on machine learning benchmark datasets.

References

[BBB+10] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, 2010.

[BNVB13] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 06 2013.

[DHK13] Li Deng, Geoffrey Hinton, and Brian Kingsbury. New types of deep neural network learning for speech recognition and related applications: An overview. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8599–8603. IEEE, 2013.

[DYDA12] George E Dahl, Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *Audio, Speech, and Language Processing, IEEE Transactions on*, 20(1):30–42, 2012.

[Hin07] Geoffrey E Hinton. Learning multiple layers of representation. *Trends in cognitive sciences*, 11(10):428–434, 2007.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[Oli07] Travis E Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.

[TH12] T Tieleman and G Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4, 2012.