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.

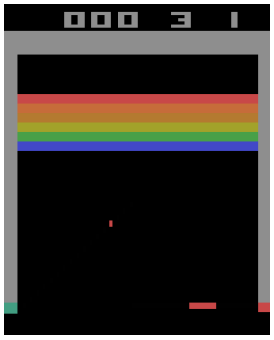

Figure 1: A game screen of Breakout.

1.2 Reinforcement learning

In a reinforcement learning setting, an agent takes actions in an environment with the goal of maximising a cumulative reward. Ee 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*.

The system learns, i.e. improves the accuracy of its predictions, by updating the weights of connections between nodes in the neural network.

[needs expansion]

gradient descent blabla

2 Components of the system

here goes the detailed description of everything (aka the hardest part)

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 Something else

...

2.5 And once again how all those things are put together

3 Implementation details

Programming language: Python 2.7 (32-bit).

Python libraries used: Pillow, NumPy, Theano.

ATARI emulation environment: ALE (Arcade Learning Environement).

3.1 Agent

...

3.2 Memory

...

3.3 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 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 2.

Figure 2: A preprocessed game screen of Breakout.

3.4 Neural network

3.5 Any other implementational stuff which is worth describing

...

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

[3] http://www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grayscale/

4 Results

Our system rules!

4.1 Performance measures

4.2 Comparison to human player

...

4.3 Comparison to the original paper

...

4.4 Something else

...

4.5 Applications and future blah

...

Appendix A: Running instructions

to test the system you should do this and that

you will see that blabla

this will make you happy

also go to github read wiki/code there for more details

References

[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.