report

July 6, 2016

1 Capstone Project

1.1 Machine Learning Engineer Nanodegree

1.2 Definition

1.2.1 Project Overview

For this project we will be creating an agent that can beat a hard coded CPU player in the classic Pong game directly from raw pixels. This is particularly challenging because the agent would have to read the game from raw pixels, and this creates a large amount of states that has been the liability of reinforcement learning.

1.2.2 Problem Statement

The following tasks will have to be completed before this project can be called successful.

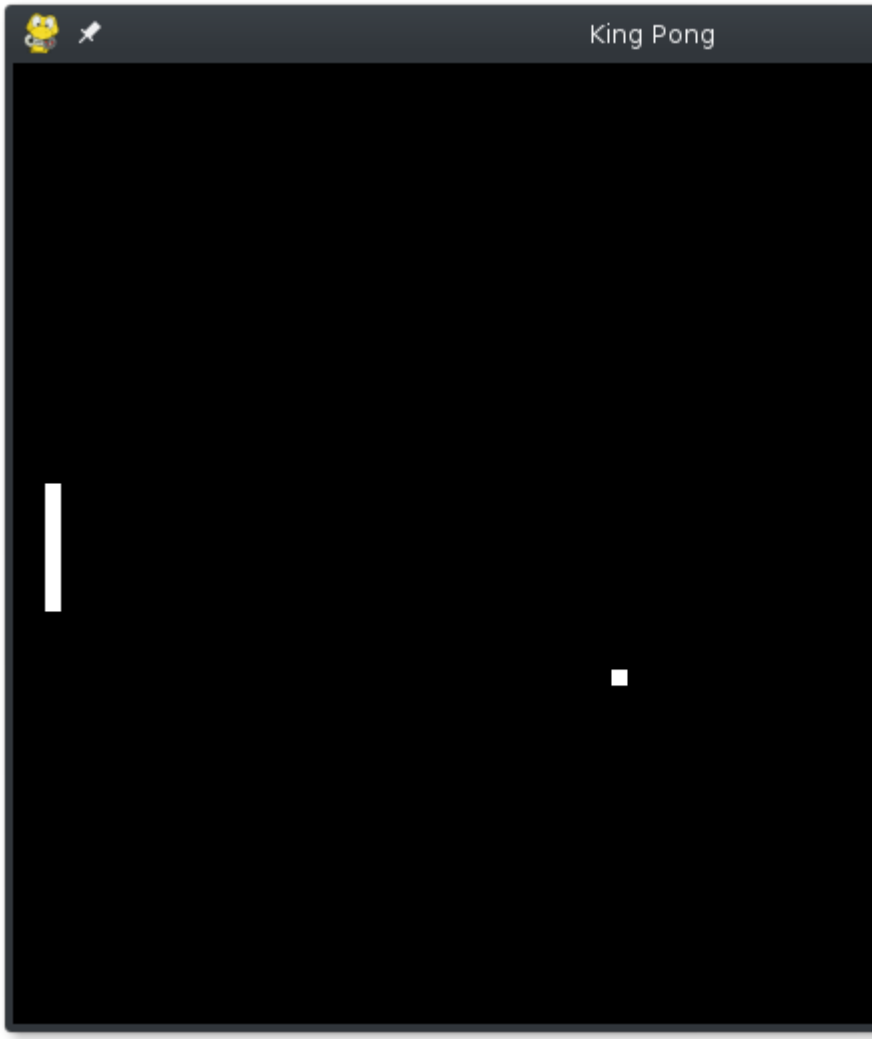
- First we need to create a Pong simulator.
- We will need to provided a hard-coded CPU player to use for our base comparisson.
- We have to read the raw images from the UI.
- Additionally, we will have to create a Deep Neural Network that can interpret and extrapolate efficiently for unseen game states.
- Lastly, we need to implement the deep reinforcement learning agent to be able to be competitive against a hard coded CPU player.

1.2.3 Metrics

To be more precise we will have to come up with a rating for the agent. Simply, we will match the fully trained player against the hard-coded CPU, stopping the match when **the first player reaches 100 games**. Each game will be 3 points. That is, the first to score 3 points wins a game, the first to win 100 games is the **King**. Our goal is to get to the 100 points first, but hopefully we can keep the CPU player under 70 points.

1.3 Analysis

For this Deep Reinforcement Learning problem there is no dataset per se, but there is, obviously, that which the agent will consume.



This is how the game of King Pong looks like:

1.3.1 Data Exploration

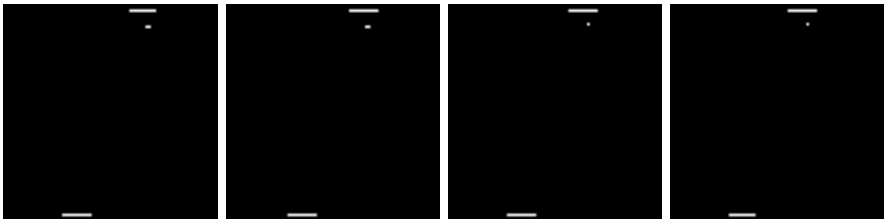
The game is a pygame window of 640 by 480. The agent reads the window using OpenCV into an array and then shrinks the image down to a 80 by 80 array. Though the shrinking also causes resolution distortion for a human looking at the images, the agent is able to extrapolate and find patterns that help him generalize into the correct states.

Not only the image is resized but also compress into a grayscale image. Think about it, although the image will read color channels, it is important to reduce the input space as much as possible. One way is to get the greyscale version of the image read.

However, you might be wondering, but how does an image tell me the direction in which the ball is going? This creates an additional challenge. To solve this problem we expanded our input array to contain 4 of this images at a time. The sequence of images does indeed tell you the direction and speed in which the ball is traveling.

1.3.2 Exploratory Visualization

This is a sequence stack of images. This is what our agent receives as an input space:



1.3.3 Algorithms and Techniques

For this project we will implement a variant of the Deep Q-Learning algorithm that the Google Deep Mind team release a couple of years ago.

For the input space we will implement a Deep Learning Neural Network using TensorFlow. This Deep Convolutional Network will take care of receiving the input image stack and extrapolating to find the best of 3 moves (No action, Up or Down.)

The first layer of the network will have 80 by 80 by 4 to be able to get the input of 4 images 80 pixels wide by 80 pixel high. Then, we connect this layer with a convolutional layer that reduces the input further to 8 by 8 by 4, and create a densely connected layer of 1600 connections into 512, and finally we connect these to a readout network with 512 inputs and 3 outputs.

This network cost function will be the square mean error and we will be training the network with an Adam Optimizer which is known to be an algorithm that converges faster than the generic Gradient Descent.

For the Reinforcement Learning agent, we will implement a version of the Q Learning algortihm. That is at it's core, it is just the same we implemented in Project 4, but this time, the learning of the agent will have a period in which it only collects information. That is, the agent doesn't train in order to allow it to collect several samples. After approximatelly 50,000 steps, then we allow the agent to change the networks values.

Also, we will be using an initial epsilon value of 0.6 and decaying it to about 0.01 over the course of several steps. This will allow the agent to explore random moves every time we restart training and converge to an almost optimal policy in the end.

1.3.4 Benchmark

This project doesn't have an obvious benchmark. However, we will be using human performance collected from average players to compare the agent. Additionally, the agent will be playing against the hard-coded CPU player which needless to say, it is a very good player. Want to try? It won't take even 2 minutes.

For a best of 3 games of 2 points each, run:

```
python king_pong.py
```

I bet the CPU wins.

1.4 Methodology

There were several steps that we took in order to make this work. The Deep Network was a challenge, and the learner took a long time to show progress. In fact, we had to source a GTX 980 and install CUDA support on our server in order to iterate over our implementation fast enough.

1.4.1 Data Preprocessing

As discussed above, the data wasn't preprocessed once, but the implementation includes a preprocessing routine that does the job continuously. Again, from a 640x480x3 read of the screen down to a stack of 4 x 80x80 grayscale images.

In addition to that, the scoreboard that can be seen on the version of the game for humans had to be removed in the agent version. This is because the score creates some noise in the data that might not be worth looking at. For example, going down to reach the ball has very little to do with the score going 3-2, we still want the agent to go for the point.

1.4.2 Implementation

The implementation took several weeks, and we found immense help with previous solutions to similar problems, as well as prominent papers, documentation and tutorials on the diverse technologies we had to merge. For the deep neural network the Udacity course and tutorials were helpful, and for the reinforcement learner the Reinforcement Learning class and a previous implementation of a similar agent FlappyBird. Also, for the actual game of Pong we started with an implementation found on the web, but quickly noticed that we had to rewrite the most of it, so it was done.

1.4.3 Refinement

Several refinements had to be made. First of all, the agent had to train for several days to reach a good performance. Also, having many hyperparameters, we iteratively chose values that made sense for the problem. The AdamOptimizer, for example, when set with a learning rate too large (10-4) will learn suboptimal policies, in the end, a value of 10-10 was selected. Also, for the gamma value 0.90 worked best.

1.5 Results

The results of this project are remarkable. Not only the agent was able to beat the benchmark but in fact the agent was able to beat the CPU player on a 100 games matchup consistently. By very little, but still it did win.

1.5.1 Model Evaluation and Validation

For an observation of our results, please, enjoy the agent on a 3 points per game 10 games match up against the CPU player.

```
python agent.py -g 10 -m 3 -vv
```

This should only take about a few minutes.
Concretely, we run the above command 3 times; the agent was able to win:

- 10 of 19 games
- 8 of 18 games
- 10 of 17 games

The human player, on the other hand won:

- 0 of 3 games
- 2 of 5 games
- 3 of 5 games

The tournament against the human player was limited to 3 games because on a 10 games the performance would drop considerably. Further comparison is shown next.

1.5.2 Justification

We argue the agent is a very solid solution to the pong game. Though, we weren't able to limit the CPU player to our desired 70 games on the "first to 100" tournament, the agent was able to win 100 out of 189. Also, when playing against the CPU, the average human player gets less than 50% the games against the CPU on a 10 games match. On the other hand, the agent often wins the most amount of games.

To show numbers we ran 3 times 100 games of 3 points, the results were as follows:

- First iteration CPU won 100 to 98
- Second iteration Agent won 100 to 93
- Third iteration Agent won 100 to 94

This can be considered close victories, and they are, but this is also related to the training time. Remember, this is an agent that got his first game after the 5,000,000th timestep. The improvement would just take some extra training time on the NVIDIA GTX 980. And it is in fact training as I write these lines.

1.6 Conclusion

This was a very interesting project we got into. Not only we applied a Machine Learning technology, but in fact 2 of the most prominent technologies as of 2016. We proved how hard it is to do Deep Reinforcement Learning, but also how satisfactory it is to get great results. We would be doing more of this kinds of work in the near future.

1.6.1 Reflection

Since we used cutting edge technologies, some of the steps we had to take in order to make this project work were completely in the dark. We had very little guidance that if we had picked some other technology, it would had been much easier to follow examples. For example, deep learning is a very new technology, with the emerging tools such as TensorFlow there is enough material in how to process images, but little in how to process a series of images as to model the motion of the ball. This challenge was resolved when reading very useful posts about deep reinforcement learning, but the practical examples were not the norm.

1.6.2 Improvement

We can think of some improvements the agent would benefit from. For example, currently we only sample 60 images from the 50,000 images in the memory queue. This is a rather small portion of what could be sampled. Also, currently there is an equal probability of sample frames that conveyed a reward than frames that were meaningless to the game.

For a next iteration of this agent, we will be implementing a more complex learning routine that would take into account those features and probabilisticly select frames that are to be of meaning to the agent.

Lastly, training time, which with a Nanodegree rolling could be expensive. We will, however, further improve the agent and make its latest version available on GitHub soon.

1.7 About running the program

1.7.1 Installing dependencies

At least the following dependencies were installed to complete this project:

- pygame
- tensorflow
- opencv2
- shapely
- numpy

Use your preferred method for installing these.

1.7.2 Running the scripts

For more information about the scripts, please run the help on the agent.py:

```
python agent.py -h
```

And if you want to play against the CPU just, as mentioned above, run:

```
python king_pong.py
```

In the future we will add a switch and code to allow a human to play an agent, and other agents to play eachother.