Invictus Capital

Data Scientist Assessment

Game-playing agent

Hannes Engelbrecht

May 2020

GENERAL OVERVIEW AND CONTENTS

This report details the work done for the second Data Scientist assessment – create a game-playing agent that can play single-player Pong. The agent is trained from zero-knowledge in an unsupervised fashion.

The following areas are discussed:

1. External Python **libraries** used in the implementation
2. General overview of the **approach** taken, and **custom rules** incorporated into the game environment and architecture
3. A complete overview of the game-playing agent's **architecture**
4. The process of **training** the game-playing agent
5. A brief overview and visualization of the agent's **performance** while it trains
6. **Bonus questions**

The aforementioned steps were conducted in a Jupyter Notebook. In addition, the trained model and its parameters were saved separately and made available on Github (https://github.com/HannesEngel/Invictus_Capital_Pong_Agent) for the project reviewer to reproduce the results. Instructions on how to do so can be found in the provided Github link's README.md file.

1. LIBRARIES

Below I have set out each of the external libraries, along with a brief overview of its main uses, utilised during the course of this project.

- NumPy – for working with arrays
- Matplotlib – for visualizing the agent's performance
- Json – for loading the JSON model file (in which the trained model is stored)
- Tensorflow – open source machine learning (ML) platform, particularly useful for building more complex ML models
- Keras – neural network library (built on top of Tensorflow) with a high-level API. Keras was used to construct and train our agent and makes use of a Tensorflow backend.
- OpenAI Gym – a toolkit for developing and comparing Reinforcement Learning algorithms. It supports teaching agents everything from walking to playing games like Pong and Pinball. It's documentation can be found here.

2. GAME-PLAYING AGENT: OVERVIEW OF OUR APPROACH AND CUSTOM RULES

For this project I have selected single-player Pong and therefore only one game-playing agent is trained. The idea is to teach the agent to play single-player Pong from 'scratch' (no preconceived notions about how Pong works) and without being explicitly programmed (no rules are hardcoded into the agent). The learning process is only guided by a reward: -1 after missing the ball, +1 if the opponent misses the ball.

For this purpose, we will make use of Reinforcement Learning (RL) to train the model, which is an increasingly popular machine learning (ML) method for this type of problem. RL has proven capable of teaching agents human-level performance on a wide variety of ATARI and other games. For example, DeepMind's AlphaGo was the first computer to defeat a professional human Go player. Similarly, IBM's Deep Blue was first computer chess-playing system to win both a chess game and a chess match against a reigning world champion under regular time controls.

Furthermore, we combine the RL with Deep Learning. That is, we combine a neural network (hence the association with 'Deep') and RL to facilitate the learning of the agent. This is done by creating a neural network in Keras that, at each step, recommends to our agent whether it should move up or down based on what it has seen in the past.

Although we are creating a single-player agent, our agent will still have an opponent. The opponent is an ATARI 2600 AI agent ('AI agent') from the gym[atari] library. This AI agent has prior training and follows (and returns) the ball relatively well.

Keywords:

- Step: Consecutive generation of a pixel frame, from one step to the next
- Point: Awarded to a player after getting the opponent to miss the ball (+1 point) or after missing the ball (-1 point). A single point consists of multiple (perhaps dozens) of steps
- Game: Series of frames until a player is awarded 1 point
- Episode: Series of games up to the first player achieving a total of 21 points
- Action: The process of our agent selecting UP or DOWN – only two actions are available for selection
- Agent: our own custom, RL agent being trained
- AI agent: ATARI 2600 opponent

3. GAME-PLAYING AGENT: ARCHITECTURE

Environment:

Our agent is trained using the pixels on the screen. Images of shape 210x160x3 (3 pixels for the RGB values) are generated and used as input to the training process. The images are however pre-processed before being passed to the model for training. We set up the pixel environment by making use of the gym library and pointing to the ATARI 2600 environment (env = gym.make("Pong-v0")).

Play is conducted horizontally, with our agent on the right side of the screen and the AI agent on the left side of the screen. Both our agent and the AI agent can only move their bats either up or down.

The ball speed is determined by the gym environment and has not been set explicitly.

Agent:

Our agent is a 2-layer fully-connected neural network constructed in Keras. The hidden layer contains 200 units and has an input dimension of 80*80 (=6400). The hidden layer has a ‘relu’ activation function, usually a good first choice for a neural network activation function. This activation function returns can be mathematically expressed as y = max(0, x). The neural network’s weights are initialized using a Glorot uniform initializer, also called the Xavier uniform initializer.

The output layer contains a single neuron and a ‘sigmoid’ activation function, which transforms the input to that layer into a value in the [0, 1] range. This is a suitable selection seeing that our model predicts the probability that the next move should be up or down (with only these two choices being available). Its weights are initialized using the RandomNormal initializer, where initial weights are generated from the normal distribution.

We also specify that the model needs to make use of the ‘binary_crossentropy’ loss function when its accuracy is evaluated during the training. This essentially boils down to the model being able to select either one of two options: move up or move down. We are also required to select an optimizer when fitting the model. This parameter allows us to determine the appropriate learning rate for the neural network. We have chosen an adaptive learning rate ‘adam’ that computes individual learning rates for different parameters. It is considered a ‘state-of-the-art’ optimization algorithm in recent ML history.

4. GAME-PLAYING AGENT: TRAINING

As mentioned, our agent learns through a very iterative approach. One could argue that RL, on the face of it, is a simple process. Our agent plays the game (through the environment that is set up) and observes the outcome. If the outcome is positive, update the model's parameters such that it encourages similar behaviour in future iterations. If the outcome is negative, update the model's parameters to discourage similar behaviour in the future. However, there's a lot more going on under the hood of this process…

Below is a slightly more detail explanation of the process followed during the training of our agent:

- Step 1: We create a random pixel frame of size (216, 160, 3). This is done through utilising the gym library we imported earlier.
- Step 2: In this step the frame we created in step 1 is pre-processed. In particular, we create a 1-dimensional vector of shape 80*80 (=6400). This is done through the use of Karpathy's 'prepro' function, which he used to code a Pong-playing agent in 130 lines of Python code.
- Step 3: We create a vector 'x' – the difference between two consecutive steps (which are actually pre-processed pixel frames). The reason for this is to try and 'detect motion' between steps.
- Step 4: In addition, a vector 'y' is created as well. Entries in this vector denote the action taken at each step, up or down. Initially, this vector will be empty seeing that no previous up or down actions will have been taken.
- Step 5: We use the model to predict the next step we should take; up or down. At this point the model provides us with two probabilities – one for choosing up, and one for choosing down. We sample from this distribution and select the appropriate action accordingly.
- Step 6: The above steps are repeated until we reach the end of an episode, i.e. where a player is the first to reach 21 points. Once this occurs, the following happens:
    - Firstly, a reward is assigned to the action chosen in step 5 above. This reward can be -1 (agent misses the ball), 0 (nothing happens; ball is in-between the two players), +1 (opponent misses the ball). The episode ends automatically, when either our agent or the opponent amasses 21 points.
    - Once the episode is done, it's time to update our model. This is where we retrain our model, using the difference between consecutive frames as input and the actual actions as the labels/output. This is where our model learns – parameters are adjusted up or down, trying to encourage those moves that lead to the highest rewards. If an action leads to a positive reward, it tunes the weights of the neural network so it keeps on predicting this winning action. Otherwise, it tunes them in the opposite way. All actions taken in the previous episode are used in this step.
    - When the model is retrained, we need to specify the 'sample_weight' parameter. For this, we make use of Karpathy's second function 'discount_rewards'. This function allows our agent

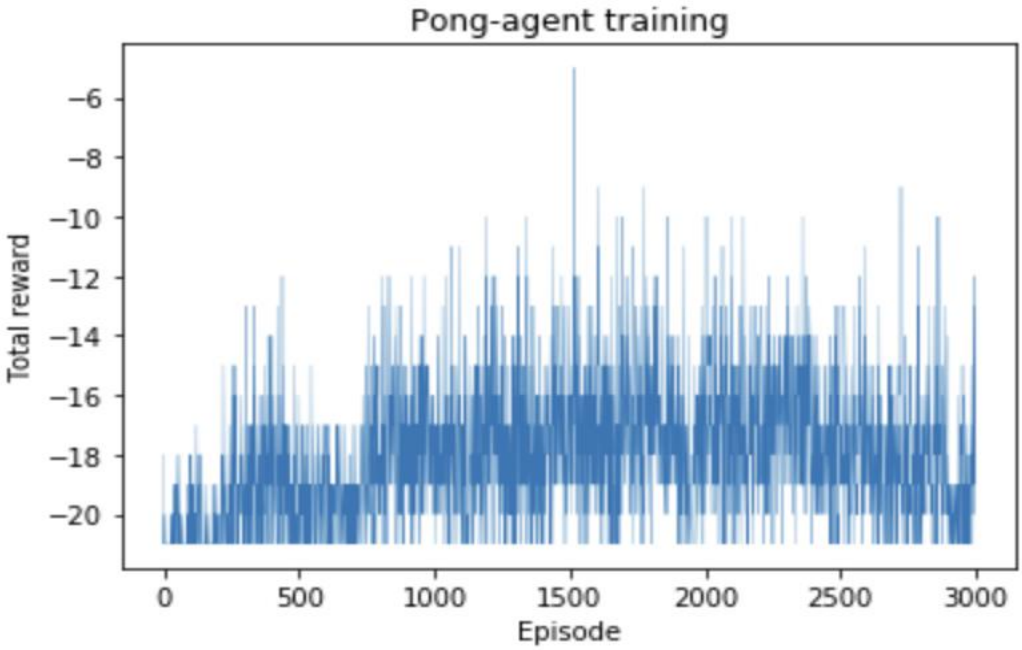
to do better by being smarter in the way we allocate a reward for an action through reward discounting. The most recent action or frame prior to a reward being received by the agent is the most relevant and therefore should be encouraged in the case of a positive reward and discouraged for a negative reward. Any actions or frames further back in time from when the reward was received are attributed with less of the credit by an exponential factor discounting factor, *γ (gamma)*, which is initialized to 0.99 ().

- We reinitialize our environment and variables before the next episode kicks off. Among the items reinitialized are the x labels (input to the model), y labels (output of the model), the history of rewards (cleared so we can keep history of the next episode's steps) and the pixel environment. Note that when the pixel environment is reset it keeps track of the previous step's state, thereby allowing the ball to follow a natural (straight trajectory).
- Next episode is initialized.

We trained our agent for 3000 episodes, each time becoming more aware of what it needs to do to win at Pong. This took approximately 12 hours to run on my Macbook Pro 2017.

5. GAME-PLAYING AGENT: PERFORMANCE

During our agent's 3000 rounds of training it was unable to win a single episode against the AI agent. Our agent was able to score only several points per episode on average. Upon investigating our agent's performance closer, we see that the most points it scored during training was 16. The graph below represents our agent's rewards across all 3000 episodes. For example, if our agent's score for a particular episode is -18, it means that it scored 3 points during that episode, whereas the opponent scored 21 points during that episode (-21 + 3 = -18). From the below we can see that our agent was unable to be the first to 21 points in all of the trained episodes. The white bars represent those episodes where our agent was unable to score a single point in the episode.



Another point to note from the above is that the agent's performance improves over time (upward trend from left to right), albeit relatively slowly. More training is required before the agent will be able to equal/beat the AI agent. Many thousands of additional episodes could aid our agent to become a winner!

Additionally, no hyperparameter tuning was conducted during the training of our agent. Another way of improving the performance of our agent could be to tune our parameters, for example the number of neurons in the hidden layer or the total number of layers in the network. I would however recommend doing so when significant compute power is available, e.g. a GPU on AWS.

6. BONUS QUESTIONS

Questions 6.2 – 6.5 were answered as part of the Language Detection assessment and can be found in that assessment's report.

6.1. ALTERNATIVE ML APPROACHES

1. We could use a similar method to the one taken in this project (Policy Gradient method), but using the RAM state of the ATARI AI agent as input to the model as opposed to using the screen images as input. Atari 2600 has only 128 bytes of RAM. On one hand, this makes our task easier, as our input is much smaller than the full screen of the console. On the other hand, the information about the game may be hard to retrieve.

For this approach we could still make use of a Keras model. One suggested architecture for such a model includes 2 hidden layers with 128 nodes each and 'relu' activations.

2. Cross-Entropy Method (CEM)

To beat our AI agent, one algorithm we could use is called the Cross-Entropy Method. The CEM is a simple evolutionary algorithm where sets of parameters are sampled from a multivariate gaussian distribution. Each sample has its corresponding policy, which determines the action our agent should take given a set of parameters. A training episode is conducted using each of the different parameters sets. The samples which earn the highest rewards are kept. It then takes these best samples and uses them to calculate a new mean and standard deviation, to be used by our gaussian distribution in the future. By doing this, our best parameter sets push the direction in which new ones are sampled in the future. Steps involved in this method can be summarised as follows:

- Step 1: Draw a bunch of initial weights from a random distribution. Although this distribution is generally chosen to be Gaussian, you can choose any distribution that you believe the weights are from.
- Step 2: Let the agent pick actions from the policy network based on these weights, run the agent through an episode and collect the rewards generated by the environment.
- Step 3: Find the weights which generated the best rewards. In general, you consider best *n* weights, where *n* is chosen by you.
- Step 4: Pick the new weights from a distribution defined by the *elite* weights.
- Step 5: Repeat steps 2–4 until you the agent has trained satisfactorily.

6.2. SUPERVISED VS UNSUPERVISED LEARNING

6.3. CLASSIFICATION VS REGRESSION

6.4. CLASS IMBALANCE: CONSEQUENCES AND SOLUTIONS

6.5. DEEP LEARNING VS NON-DEEP LEARNING APPROACHES

6.6. SUGGESTIONS: DYNAMIC DIFFICULTY ADJUSTMENT BASED ON PLAYER PERFORMANCE

One suggestion for adjusting the difficulty for the human player would be to vary the speed of the ball depending on the current score. For example, if the human player gets ahead by a certain number of points, e.g. 5 points, the speed of the ball could increase. If the player gets ahead by 10 points, the ball could accelerate even more, and so forth. Once the score 'normalises', the ball speed could reduce again to make it easier for the human player. This would however require us to train our agent at varying ball speeds to ensure that it is itself able to cope with increasing ball speeds.

Another suggestion would be to add some randomness to the agent's chosen actions and to let it make deliberate mistakes. This is referred to as 'artificial stupidity'; therefore, the player may remain interested in the game for a longer period of time, The catch is, however, to ensure that the mistakes are not too obvious.

7. REFERENCES

- Andrej Karpathy, Learning to play Pong from Pixels by Andrej Karpathy
- Michael Trazzi, https://blog.floydhub.com/spinning-up-with-deep-reinforcement-learning/
- Omkar Vedpathak, https://towardsdatascience.com/intro-to-reinforcement-learning-pong-92a94aa0f84d