Playing Pong

Abstract

The project is about learning the AI agent to play the Pong Atari game, which is an imitation of table tennis. Reinforcement learning techniques are used to train neural network models with help of the Python libraries such as Keras, Tensorflow, Numpy and OpenAI. The goal is to demonstrate step by step how such a model can be constructed. The explanation of theory as well as code snippets are presented. The model's performance is discussed and the conclusions are presented.

Course name: Reinforcement Learning

Course of study: Artificial Intelligence

Date: 07 April 2022

Author: Bartlomiej Palus

Matriculation number: 92017197

Tutor: Prof. Max Pumperla

Contents

List of Figures

List of Equations

List of Tables

List of Abbreviations

NN – Neural Network

RL – Reinforcement Learning

1. Introduction

Reinforcement learning is a branch of machine learning in which the so-called “agent” and “environment” can be distinguished. The interaction of the agent with the environment(called an “action”) is essential and during this interaction, rewards are collected by the agent. The goal is to maximize these rewards. It can be done by a trial and error approach in which the exploration vs exploitation dilemma appears. It is the problem of either choosing to explore the unknown environment more and collect more information about it to make better decisions in future or using what information has been collected so far and tends to maximize the reward on its basis. It is up to the developer to appropriately adjust it.

This paper is a hands-on project that aims to explain step by step implementation of the Python code that builds a model that can learn how to play Pong through a reinforcement learning approach. Several Python libraries like Numpy, Tensorflow, Keras, OpenAI etc. are used to achieve it. Neural networks are employed to construct a reliable and efficient model. To present the code *italics* is used.

Pong is a video game that resembles table tennis that was released in 1972 by a company named Atari (Wikipedia, 2022). It has simple two-dimensional graphics. There are two paddles at each end of the screen and a ball that is supposed to be bounced off from these paddles by players. There are two players and each of them controls one of the paddles. They compete with each other and the one that makes the other miss the ball gains a point. The game is played until the specified point limit is reached.

Such an easy game setup is very good for performing a reinforcement learning analysis on it. The agent is controlling the paddle. The environment is the space in which the game is playe. There are three actions - the paddle can:

- Remain stationary
- Go up
- Go down

Although there are three actions possible in the Pong game the probabilistic model employed in this problem will consider only two of them – going up and going down. This is so because the agent is assumed to be in movement to learn and play more efficiently.

One can wonder why is reinforcement learning needed in such an easy setup with only three possible actions. Although it may seem an easy task to develop a game like this with a programmed player that plays the game perfectly it is not so. The common Pong game emulator outputs 210x160x3 display which is 100 800 pixels (Mnih V, Kavukcuoglu K, Silver D, Graves A, Antonoglou I, Wierstra D, Riedmiller M., 2013). Although, many of them are not relevant in the case of the pure “programmatic” approach, even if 50% are get rid of it is still a lot of them. If it was to be manually programmed, it would come to coding each possible state and action. This would create an enormous amount of data and engage a lot of developer’s time. Even if this was performed, it would be a very Pong specific game and each move would depend on the developer’s creativity and the actions would be hardcoded. Such an opponent would be very susceptible to new, unpredicted actions. The reinforcement learning approach lets the agent learn how to play and improve with each epoch of the training. This makes it very versatile. The developer’s job is to only properly create a model and let the algorithms do the job.

The way in which this RL approach works is basing on pixels. Instead of hard-coding the actions that computer-controlled player can perform the AI model learns from the position of paddles and a ball, which it reads through pixels. It is much more like humans learn how to play. We observe the action in the game and adjust our response to it. This is the huge power of reinforcement learning.

2. Setting up the programming environment

It is a good practice to set up an environment for the new project. Here, python programming language is used. Anaconda is employed in order to take help set up the environment and compile the code.

Following command are used in anaconda command prompt to create the appropriate environment:

```
conda create -n pong numpy tensorflow keras jupyter notebook matplotlib
pip install --upgrade pip –user
pip install gym
pip install gym[atari]
pip install ale-py
```

This allows all of the python libraries to work correctly without breaking the dependencies.

3. Programming

Maths in reinforcement learning is crucial to understand how the machine learning libraries work and to comprehend the concepts. But the process of transforming that math into code gives the visible results in the form of an application that can be used by a user who does not necessarily understand the algorithms. This is the power of programming, and this section will focus on it.

3.1. Importing libraries

First of all, it is necessary to import the previously downloaded libraries to the coding environment. This is quite a lot of code, so instead of listing them, it suffices to mention that Keras, Numpy, Tensorflow, OpenAI and matplotlib are imported.

3.2. Creating the Pong environment

The OpeanAI gym library allows creating environments for different Atari games. In this case, it is an instance of Pong.

```
env = gym.make(“ALE/Pong-v5”)
observation = env.reset()
prev_input = None
```

3.3. Defining agent’s actions

For each reinforcement learning project, it is necessary to define actions that the agent can perform. In Pong game, it is be able to move up, down or stay still. The action to remain still is pre-defined in *gym* package, so there is no need to code it up.

```
ACTION_UP = 2
ACTION_DOWN = 3
```

3.4. Setting up hyperparameters

To understand reinforcement learning better let’s consider one of its popular algorithms, that is Q-learning. It works based on updating the state’s value. The value defines how “attractive” taking particular action from this state is. It is represented by the following equation:

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha)\cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

Equation 1 Q-learning (Wikipedia, 2022).

The Q^{new} is the updated value. The learning rate(α) determines how much the information from the old value is retained. It can take a value from 0 to 1. When 1 is used it means that the old value does not influence the update. Only the estimate of optimal future value and the reward are the factors that influence the update. Tuning the learning rate switches the weight between those accordingly. The point here is that the implementation of the algorithm in practice is not always straightforward. In the case of the learning rate, although it is an important factor in the RL, in this project it is not directly specified. It is “hidden” in the optimizer of the loss function. If the SGD was used for example, then it could be defined as its parameter, but here the Adam optimizer is used which does not require setting the learning rate.

The discount factor(γ) has an impact on how the future reward is treated. The higher γ is, the earlier the agent adjusts its actions according to the possible future reward. The γ=0.99 seems a good choice to make because it is important that the agent focuses on moving the paddle in such a way that it will result in beating the opponent. This can be achieved when the agent anticipates what move to take many time steps before it actually is supposed to hit the ball with the paddle.

That way it has time to move the paddle efficiently. It is important to remind that in the Pong game that is built in this project one time step is a difference between two frames. One frame is defined as when the ball moves by one pixel. Taking into account how many pixels there are in the game it is clear that important decision states(in the meaning of scoring a point) are far away on the horizon. Thus, *γ=0.99* is set. Then, initialization of the training set, rewards and episodes is done.

```
x_train, y_train, rewards = [ ],[ ],[ ]
reward_sum = 0
episode_nb = 0
```

The reward for scoring a point(ball passes by opponents paddle) is defined as 1 and for losing a point(ball passes by the AI agent's paddle) is defined as -1. It is worth mentioning that by default when the ball is in motion the reward is 0 because the score does not change. This is coded inside the OpenAI gym library. In the reinforcement learning approach, the situation is different because of the already mentioned discount factor. A separate function will be built and explained later on to deal with this important concept.

3.5. Pong frame

As has been already mentioned a basic unit of data within the game space is the movement of the ball and paddles by one frame. To visualize this the instance of Pong is called:

```
env = gym.make("ALE/Pong-v5")
observation = env.reset()
```

In the code below, 25 iterations of the Pong frame are done and the last 4 frames are displayed. The result is presented in graphical form in Figure 1 Pong frames. It can be observed how does the movement of the ball look in subsequent frames. It can already be concluded that in the Pong game obtaining the reward has a long perspective. There are many subsequent frames in which the ball is just moving from one side of the screen to the other, though getting no reward. That means that the learning of the model is hampered in this aspect.

```
for i in range(25):
   if i > 20:
      plt.imshow(observation)
      plt.show()observation, _, _, _ = env.step(1)
observation, _, _, _ = env.step(1)
```

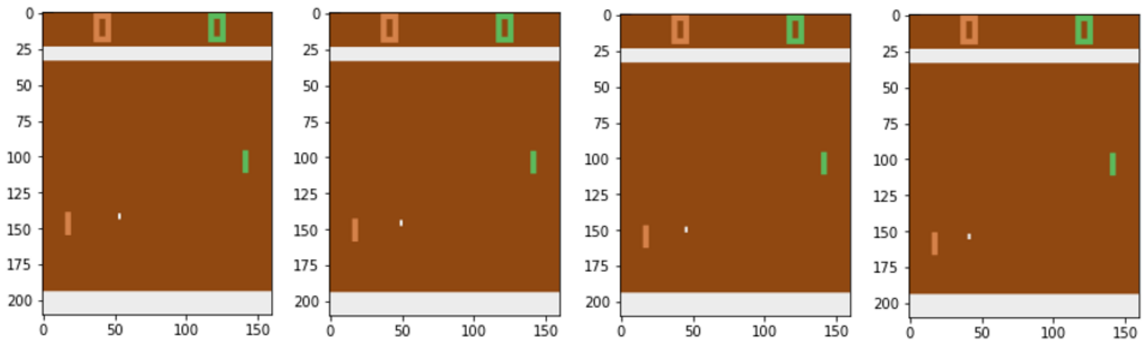


Figure 1 Pong frames.

3.6. Pre-processing

The Pong game environment in Figure 1 above, as previously mentioned, has 210x160x3 resolution. This is computationally demanding and it is a good idea to decrease it. First of all, some of the details are not important in the analysis. The scoring is done internally inside the OpenAI gym environment, so the scoreboard can be discarded. Also, the grey beams are not needed. This is done in the first four lines of the "*prepro"* function. The most important, for the sake of this analysis, are the two paddles a ball and a space in which the ball can move. The RGB is only a visual aspect, so it is degraded to grayscale(fifth line inside the function). The cropping of the 210x160x3 image results in a 6400 pixels vector coming from an 80x80 grayscale image. Such a 6400x1 column vector is constructed at the return statement of the function. The last three lines generate the cropped image visible in Figure 2.

```
def prepro(I):
  I = I[35:195]
  I = I[::2,::2,0]
  I[I == 144] = 0
  I[I == 109] = 0
  I[I != 0] = 1
  return I.astype(float).ravel()
obs_preprocessed = prepro(observation).reshape(80,80)
plt.imshow(obs_preprocessed, cmap='gray')
plt.show()
```

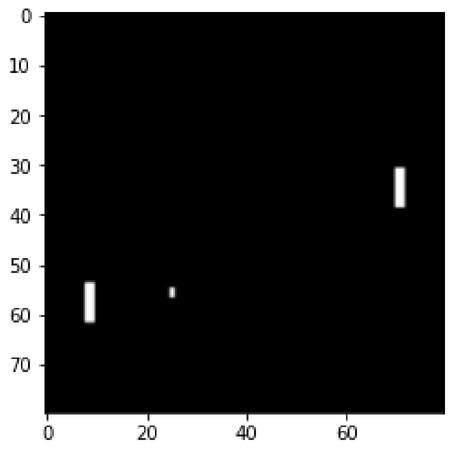


Figure 2 80x80 Pong frame.

3.7. Reward

What is fed as input to the model is the difference between two subsequent frames. As mentioned before each time the ball is in the space between the paddles the default reward is 0. In such a situation the reward vector would contain mainly zeros. This would make the predictions based on the reward very unstable. There would be no information for the algorithm about what action to take when in the, let's say, 30 steps on the horizon there would be only zeros. This is why the discount function has to be programmed. It makes use of the discount factor(γ) that has been defined earlier. Thanks to it, the future reward can influence the preceding frames and distribute its normalized form among them.

First, we construct the Numpy array from the rewards vector that is fed into the function. Then a duplicate of this vector is constructed but filled with zeros only. The “running_add” is initialized to accumulate the reward from the upcoming loop. Next, we construct the reversed loop that goes from the end of the vector to the beginning, so there is no need to do exponentiations. Then, the “*if”* condition is triggered when the game ends(we have exhausted all the collected rewards) and the sum of the rewards is reset. Inside this loop, the discounted reward vector is calculated according to the Q-learning formula. Then, the updated form of it is normalized and returned.

```
def discount_rewards(r, gamma):
 r = np.array(r)
 discounted_r = np.zeros_like(r)
 running_add = 0
 for t in reversed(range(0, r.size)):
      if r[t] != 0: running_add = 0
     running_add = running_add * gamma + r[t]
     discounted_r[t] = running_add
discounted_r -= np.mean(discounted_r)
discounted_r /= np.std(discounted_r)
return discounted_r
```

3.8. Neural network

The neural network is the thing that makes this reinforcement learning programming project so powerful. The raw pixels from the Pong game are an input to it. The output is the actions that the agent performs. Due to previously defined rewards, the NN is able to learn which actions are more favourable than others. If a particular set of actions result in the agent winning the point, this set of actions is rewarded and the weights in the neural network are updated to favour them. On the other hand, if the point is lost then the particular set of actions would be discouraged and less likely to be repeated.

To dive a little deeper into this reasoning a particular situation can be considered. Let’s assume that in a certain episode the agent bounces the ball off its pad two times and then when the ball approaches it for the third time it goes past it and the point is lost. Although the agent’s actions were correct two times the obtained reward is negative because the point was lost. This leads to discouraging all four actions and may seem counter-intuitive. But the real power of artificial intelligence is that the models can be trained thousands or even millions of times. Thanks to the huge computational power available right now it is possible. After training the model for a longer time the two correct actions will be repeated and at some point, a third random action will appear which will lead to obtaining the point. From now on, this set of actions will be favoured and the longer the model is trained it will be encouraged more and more. Of course given that the model is properly set up, with appropriate hyperparameters etc. This illustrates how powerful reinforcement learning is and what a big potential it has.

Coming back to the coding part, let’s consider how the neural network in this project is built. The Keras library is used to implement it. The sequential model is employed to stack the neural network layers.

model = Sequential()

First, the hidden layer is added to the model. The construction of Dense NN in Keras has such a form that for a first layer it requires specifying the dimension of the input layer *“input_dim”.* In this project, it is 80x80, as explained in section 3.6 the 210x160x3 resolution was cropped to 80x80. It is nothing else than just 6400 pixels that are a 6400x1 input vector. The 80*80 notation is just to remind the original shape of the Pong frame. The hidden layer decreases the size to a 200x1 vector and is coded as *“units”.*

The activation function for this hidden layer is ReLU. It takes as input the summed weights of the hidden layer and outputs them in the unchanged form if they are positive otherwise it outputs zero. It is a commonly used activation function because it has good training performance and is easy to train.

The kernel initializer defines how the weights in the NN are initialized. Here *“glorot_uniform”* is used. It draws samples from a [-x, x] uniform distribution. Where $x = \sqrt{\frac{6}{n_{in}+n_{out}}}$, n_{in} and n_{out} are the number of units in the input and output vectors (Keras, 2022).

*model.add(Dense(units=200,input_dim=80*80, activation='relu', kernel_initializer='glorot_uniform'))*

Then we add the output layer. Now only the output dimension is specified because the input is automatically taken from the previous layer. The output dimension is simply 1. It is so because the probability of taking action “up” is modelled. The probabilities must add up to 1, so the action “down” can be concluded by simply subtracting the “up” probability from 1.

The activation function employed is the sigmoid function. That way the obtained output is between 0 and 1. It makes the result very neat, because it is a straightforward probability of taking action “up”. Then calculation action “down” is just subtracting the final result of the NN from 1.

This time the weight initializer chosen is “RandomNormal”. Which is sampling a random variable from Gaussian distribution, also called the normal distribution.

model.add(Dense(units=1, activation='sigmoid', kernel_initializer='RandomNormal'))

Neural network construction in Keras has to be finished by the compiler. The first thing to be defined is the loss function. In general, the loss function in neural networks works based on taking a difference between the predicted labels and the true ones. Then the objective is to minimize this “loss” through the optimizer of choice. In the process of minimizing it, the weights of the neural network are updated accordingly by backpropagation. In the case of reinforcement learning, this process is slightly different. The main distinction is that there are no true labels in RL. Instead, the model gets feedback if the targeted action is favoured or not by the reward. Thanks to it, the samples that lead to lower loss are encouraged and the model updates its weights accordingly.

In this project, the binary cross-entropy loss function is chosen. This is so because as mentioned previously the focus is to predict the probability of the “up” action and in consequence the “down” action. This makes the problem binary, so binary cross-entropy is a good loss function for this problem.

Having the loss function chosen, now it comes to selecting the appropriate optimizer to find the minimum of this function. Stochastic gradient descent is the most famous one and is usually introduced at the beginning of the machine learning journey. Although it is a good optimizer, the field has evolved and there are now some alternatives. One of them is the Adam optimizer. In general, it has become very popular in machine learning societies. It is tempting to use because the default parameters usually perform very well. It takes off the shoulders of the developer tuning the optimizer parameters. The comparison of different optimizers’ performance can be seen below. The lower the training cost the better the performance.

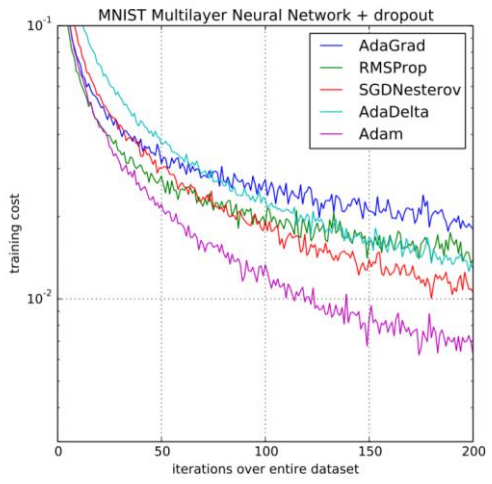
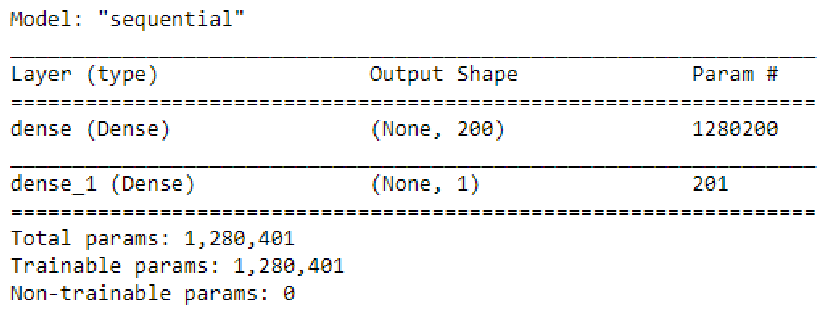


Figure 3 Optimizers’ training cost comparison (geeksforgeeks, 2022).

The last thing that is defined for the Keras compiler in this project is metrics measure. The choice is *accuracy.* It is the ratio between the number of correct predictions and the total number of predictions made. It allows checking how good the model’s predictions are during the training.

model.compile(loss='binary_crossentropy', optimizer="adam", metrics=['accuracy'])

The sum-up of the model created can be visible in the below table generated in the jupyter notebook environment using Keras library.



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 200)               1280200
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 201
=================================================================
Total params: 1,280,401
Trainable params: 1,280,401
Non-trainable params: 0
```

Table 1 Summary of the NN model.

We can see that the number of parameters exceeds a million. The first layer is significantly bigger than the second one because 200 units were set there in comparison to only 1 in the second layer. This is quite a substantial number and we can expect that a training time can be quite long in case of the computational power delivered by a personal laptop.

3.9. Training the model

As the Pong environment is set up, pre-processing and discounted rewards functions are defined and the model is prepared it is time to get it all in action. The main objective of every reinforcement learning project is to correctly train the model and obtain the best possible predictions from it. It is what the following code tries to achieve.

The *history* empty list is created to append accumulated rewards to it later in the training. Then the Pong environment is reset and assigned to the *observation* variable once again to have it clean for the training. The *"prev_input"* variable is initiated for further use as well.

```
history = [ ]
observation = env.reset()
prev_input = None
```

Now, we create the main loop to train the model. It will break when one of the players reaches 21 points, so when the game ends(players play until one of them reaches 21 points). All of the code that is going to be described in this section is inside the while loop, as below.

```
while True:
```

First, the Pong observation frame has to be pre-processed, as described previously, to decrease its resolution. It is done with the use of the function defined in section 3.6 called "*prepro*". As mentioned previously the actual input for training(called *x* here) is the difference between the observation frame and the previous frame. The "*if*" condition is introduced to get into account the first frame because there is no previous frame to subtract from it. That is why the "*if* "condition is triggered for the first frame and the vector of 6400x1(the frame has 80x80 resolution) full of zeros is defined, so it can be subtracted to avoid errors. Then previous input is assigned to the current input to maintain the loop.

```
cur_input = prepro(observation)
x = cur_input - prev_input if prev_input is not None else np.zeros(6400)
prev_input = cur_input
```

Then, the model prediction can be made. However, it should not be confused with a more common application of this predict method in supervised learning. The difference is that in the supervised the objective is to predict a correct label for a particular input data. Here, the predictions are the probability of execution of certain actions. In this project it is the probability of the paddle, being under the control of the agent, moving upwards. The previously trained model is used with the training samples *"x".* The training set undergoes a mathematical transformation. It expands the matrix and transposes it to obtain an appropriate shape for further manipulations.

```
proba = model.predict(np.expand_dims(x, axis=1).T)
```

Having the probability of the upwards movement calculated it is time to assign the labels to it. As a reminder, in section 3.3 we assigned the upwards action to number 2. Now, the probability of upwards action is compared with uniform distribution (so a random sample in range [0, 1]). In case the probability of upwards action is higher than a random sample from this distribution it means that the robot chooses to execute the upwards action. Otherwise, the paddle is moved downwards. To stick to the convention, the executed action is assigned labels either 0 or 1. 1 if it is an upwards action and 0 if it is a downwards action. Next, we assign the "x's" to the training samples and the "y's" to the training labels.

```
action = UP_ACTION if np.random.uniform() < proba else DOWN_ACTION
y = 1 if action == 2 else 0
x_train.append(x)
y_train.append(y)
```

Now it is time to use the *"step"* method on the Pong environment (we should remember that the loop is still active). It is done internally in the OpeanAI gym. Doing it gives four new variables, as visible below. The "o*bservation"* and the "*reward"* have been already explained. There are two new variables visible in the code below. The "*done"* variable will be used later on and plays an important role. It defines whether the game is over (one of the players reaches 21 points). It is also a mark of one training episode. The "*info"* variable is just for a purpose of extracting it (avoiding an error) from the step method of the OpenAI gym and plays no role in this project. Then the obtained reward (calculated as explained in sections 3.4 & 3.7) is appended to the "*rewards"* list defined earlier and also the sum of the rewards is calculated for later use.

```
observation, reward, done, info = env.step(action)
rewards.append(reward)
reward_sum += reward
```

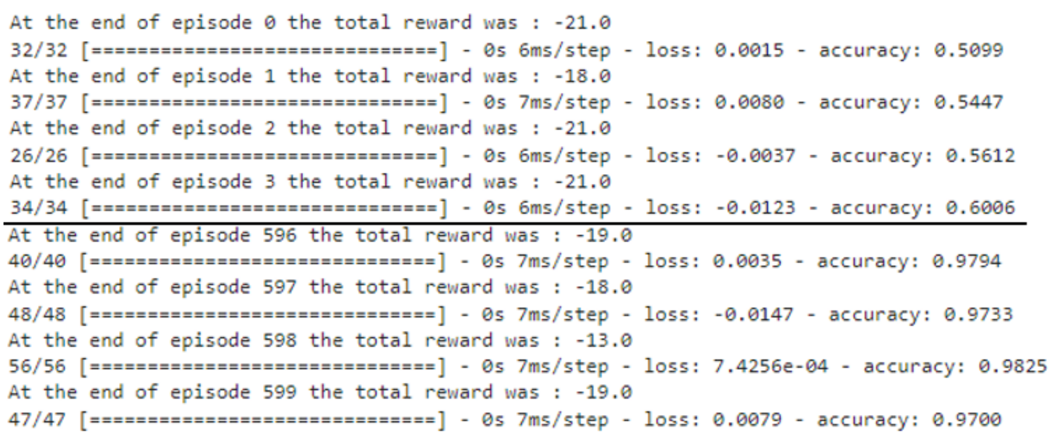
Also, inside the while loop, there is the "*if"* condition which is triggered based on a previously define *done* variable. It is activated when one of the players reaches 21 points (one game ends). Then the accumulated rewards are added to the history of all games that have been played in the training. The information is printed to the developer about how many rewards have been accumulated by the agent in which episode.

Next, the condition to break from the training loop, meaning ending the computation, is defined. It has been observed that for the agent to learn to play the Pong game on the level that it can dominate the opponent consistently, much more than 600 episode has to be computed (http://karpathy.github.io/, 2022). But for the purpose of this project 600 episode condition is set along with the accumulated reward being above -12. So the agent can learn to at least score some points against the opponent. Computing 600 episodes on the laptop take about 12 hours which is already very time-consuming.

```
if done:
  history.append(reward_sum)
  print('At the end of episode', episode_nb, 'the total reward was :', reward_sum)
  if episode_nb>=600 and reward_sum >=-12:
     break
```

The first part of the table below illustrates the first four episodes of training the model. It can be seen that the agent loses almost all balls. Rewards -21 means that in one game it missed all balls because the game terminates when 21 points have been scored. There is an instance where two points have been scored. It is so because the paddle is randomly moved to explore the environment. There exists a probability that during exploration the proper action is executed. This is why there are a few points scored at the beginning of training, but it is far from scoring the points by taking actions purposefully.

The second part of the table shows four subsequent episodes of training the model starting from the 597th. It can be seen that, although the accuracy increased significantly the reward did not increase much. There is an improvement, but the agent still losses much more balls than the opponent.



```
At the end of episode 0 the total reward was : -21.0
32/32 [==============================] - 0s 6ms/step - loss: 0.0015 - accuracy: 0.5099
At the end of episode 1 the total reward was : -18.0
37/37 [==============================] - 0s 7ms/step - loss: 0.0080 - accuracy: 0.5447
At the end of episode 2 the total reward was : -21.0
26/26 [==============================] - 0s 6ms/step - loss: -0.0037 - accuracy: 0.5612
At the end of episode 3 the total reward was : -21.0
34/34 [==============================] - 0s 6ms/step - loss: -0.0123 - accuracy: 0.6006
At the end of episode 596 the total reward was : -19.0
40/40 [==============================] - 0s 7ms/step - loss: 0.0035 - accuracy: 0.9794
At the end of episode 597 the total reward was : -18.0
48/48 [==============================] - 0s 7ms/step - loss: -0.0147 - accuracy: 0.9733
At the end of episode 598 the total reward was : -13.0
56/56 [==============================] - 0s 7ms/step - loss: 7.4256e-04 - accuracy: 0.9825
At the end of episode 599 the total reward was : -19.0
47/47 [==============================] - 0s 7ms/step - loss: 0.0079 - accuracy: 0.9700
```

Table 2 Model output

A very interesting observation is made here. Although the accuracy increases significantly while the training progresses the reward improves only slightly. The question that comes up to mind is: Why would the accuracy increase that early in the training process, since the other metrics do not seem to follow it? To understand it let's dive deeper into the theory of reinforcement learning is required.

During the training in RL the "policy" is constructed. It defines which actions are more desired in case of obtaining the reward. We can say that some kind of "instruction" is produced for the agent, for which actions are more likely and which are less likely to get the reward. Furthermore, there is the so-called "exploration vs exploitation" dilemma. It is the concept in which the agent has to choose between action that leads to getting the reward in the short-term or actions that are supposed to explore the unknown environment and give the possibility to obtain better rewards in the future.

In this project, at the beginning of training the agent is only "crawling" in the Pong space. It starts exploring the environment and builds up the policy. Although it only starts exploring it already has some "policy" built. What the accuracy is telling us, is the ratio between the agent's actions that it has already taken and the actions that seem good for the policy established so far. This is why the accuracy increases, although the agent still plays Pong horribly and does not improve significantly on scoring points.

The best measure of how well the agent plays Pong is the reward it obtains. It can be seen that after six hundred episodes of training it is still losing the game and scoring only a few points. It means that much more training is required so it can start beating the opponent.

If the loop is not terminated then the below code is executed. First, the episode is incremented. Then training samples and labels matrices are transformed to the form accepted by the fit method of the model. Next, the crucial step in the project is defined. We apply the fit method to the model. The inputs are the training samples and labels. The *"verbose"* set to 1 means that the progress bar for training is displayed. Thanks to it Table 2 had been generated.

The sample weight is an interesting parameter defined. As can be seen, it employs the previously defined function, that calculated discounted reward. To quickly remind, it calculates the rewards discounted by the gamma parameter. That way even actions that remotely lead to positive rewards are encouraged. Thanks to it a matrix of discounted rewards is created. Then it is fed to the neural networks as the weights that correspond to the training samples that these rewards are correlated with. It facilitates the process of adjusting the NN weights to favour the actions that are leading to winning the game. Then training sets, rewards, Pong environment etc. are reinitialized to start a new episode with a clean set-up. With this code, the main training "*while"* loop ends.

```
else:
  episode_nb += 1
```

```
x = np.vstack(x_train)
y = np.vstack(y_train)
model.fit(x, y, verbose=1, sample_weight=discount_rewards(rewards, gamma))
x_train, y_train, rewards = [ ],[ ],[ ]
observation = env.reset()
reward_sum = 0
prev_input = None
```

3.10. Results

The following code generates the Figure 4 with a number of episodes on the horizontal axis and rewards on the vertical one.

```
plt.plot(history)
plt.show()
```

It can be visible from this figure that the reward increases with the increasing number of episodes. As defined previously one training episode for the agent is when it is exposed to one game. One game is played until one of the players achieves 21 points. It illustrates how at the begging the agent does not know how to play and depends only on random moves. It allows it to score a point occasionally, but mainly the accumulated reward is -21. It means that all points have been lost by the agent. Later on, it improves, but it is still far from beating the opponent. Providing longer training with better computational resources it should not be a problem to train the agent sufficiently, that it starts winning the games. For this project, it is considered sufficient to demonstrate that the agent improves with the increased training time.

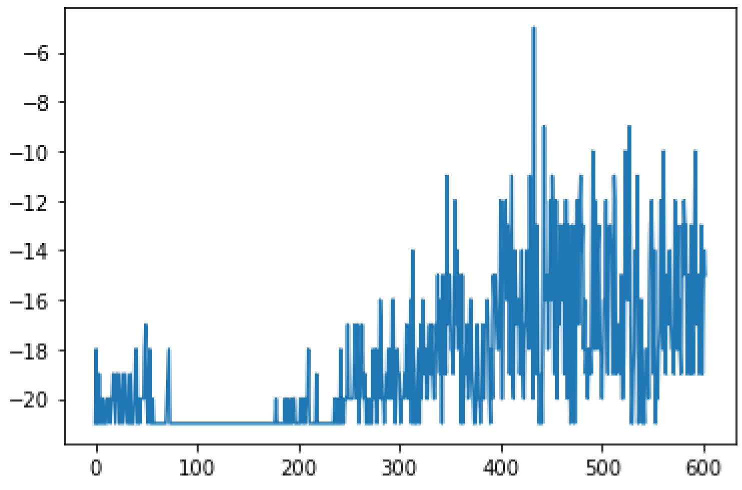


Figure 4 Reward vs number of training episodes.

4. Conclusions

Although only one hidden layer was used in the construction of the model, the computational time was very long. It took roughly 12 hours to simulate 600 episodes. It demonstrates how time-consuming the reinforcement tasks can be. After such time it did not bring satisfying results. The agent still was far from beating the opponent. Increasing the training time would improve it significantly. At some point, the agent would be able to beat the opponent and later in the training devastate him. Expanding the NN with additional layers is another idea to improve the performance, but it would also lead to elongated training time. To check these ideas a more sophisticated computational tool than a personal laptop would have to be used.

There is also room for improvement in terms of tuning the parameters of the model. For example, different activation functions can be tested and compared to achieve the best performance. Also, kernel initializers are an area in which there are more choices that could be compared. Instead of adding additional layers, more units could be added to the existing ones, but it would also lead to increased computational time. All in all, there is still much to be explored in terms of increasing the model's efficiency and expanding the research.

Reinforcement learning is a very powerful concept. In comparison to supervised learning, it does not have clearly stated true labels in the training set which allow the model to learn straightforwardly. Instead, the model learns during the exploration of the environment. As the learning progressed the agent explores more of the environment and extends its decision spectrum. However, intensive exploration does not always lead to better performance, because the unknown "territory" may not be that appealing in case of the obtained rewards. This is why the exploitation of so far learned policy is also important. It is defined as an exploration-exploitation dilemma. It is up to the developer to appropriately adjust the model and its hyperparameters to extract the best there is in the reinforcement learning algorithms.

It is worth emphasizing how big potential there is in this method. Supervised learning models have already achieved quite spectacular results in the current AI applications, but there is always a full training set for the model to be trained on. In reinforcement learning the agent learns by itself. It is an incredibly potent idea. In this project, it has been demonstrated on the example of a quite primitive game for nowadays standards – Pong. But we have done it deliberately. Using easy examples is a good idea to explain complicated concepts. It makes it clearer and more engaging.

Although, the Pong game is an old one it is still sometimes played, because of its ageless and captivating style. It is impossible to present a recording of the game played in this paper, but it is a good idea to see how such a programmed AI agent would look like in action. It is possible to do it in the OpenAI library. Some additional coding would have to be done, and the actual game-play of the programmed agent could be performed. This is a really interesting idea and it is highly recommended to try it.

Bibliography

geeksforgeeks. (2022, March 30). Retrieved March 30, 2022, from geeksforgeeks: https://www.geeksforgeeks.org/intuition-of-adam-optimizer/#:~:text=Adam%20optimizer%20involves%20a%20combination,minima%20in%20a%20faster%20pace.

Github. (2022, April 4). Retrieved from Github: https://github.com/thinkingparticle/deep_rl_pong_keras/blob/master/reinforcement_learning_pong_keras_policy_gradients.ipynb

http://karpathy.github.io/. (2022, April 1). Retrieved March 31, 2022, from http://karpathy.github.io/2016/05/31/rl/

Keras. (2022, March 28). Retrieved March 29, 2022, from Keras: https://keras.io/api/layers/initializers/

M. G. Bellemare, Y. Naddaf, J. Veness and M. Bowling. (2013). The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research, 47*, 253-279.

Medium. (2022, April 3). Retrieved from Medium: https://medium.com/gradientcrescent/fundamentals-of-reinforcement-learning-automating-pong-in-using-a-policy-model-an-implementation-b71f64c158ff#id_token=eyJhbGciOiJSUzI1NiIsImtpZCI6IjcyOTE4OTQ1MGQ0OTAyODU3MDQyNTI2NmYwM2U3MzdmNDVhZjI5MzIiLCJ0eXAiOiJKV1Qi

Mnih V, Kavukcuoglu K, Silver D, Graves A, Antonoglou I, Wierstra D, Riedmiller M. (2013). Playing Atari with Deep Reinforcement Learning.

Wikipedia. (2022, March 20). Retrieved March 22, 2022, from Wikipedia: https://en.wikipedia.org/wiki/Atari

Wikipedia. (2022, March 23). Retrieved March 23, 2022, from Wikipedia: https://en.wikipedia.org/wiki/Q-learning#:~:text=Q%2Dlearning%20is%20a%20model,and%20rewards%20without%20requiring%20adaptations.