Using Reinforcement Learning to Create a Forex Trading Bot

Marcél Busschers, Alejandro Behrens, Juan Rodriguez, Daniel McHugh, Daphne Zindili

March 26, 2021

Abstract

The objective of this research paper is to build an autonomous foreign exchange (FX) trading agent that can learn how to trade and generate profits using a deep reinforcement learning algorithm named Proximal Policy Optimization (PPO). The PPO model implemented for this research is a type of Epsilon-Greedy Deep Q-Learning model, given that the model's rewards are calculated after each step in the custom environment. Our results show that after the model has been trained several times in different environments, the bot has learned enough to understand what action should be next: buy, sell or hold. Therefore, the hypothesis aims to answer whether PPO can be an alternative when creating trading bots and whether the bot will be able to maximize returns.

1 Introduction

The rapid development of hardware and machine learning algorithms has given rise to the field of automated stock trading. As a result, several machine learning approaches have been applied in the search of successfully predicting stock prices and performing stock market analysis. Specifically, support vector machines (SVMs) and Neural networks have been some of the most commonly chosen models for these tasks [14]. More recently, reinforcement learning algorithms seem to be the most popular to develop trading agents that can learn the market's behavior and trade autonomously [2, 29, 30]. Given this trend, the aim of this paper is to build a foreign exchange (FX) trading bot based on a deep reinforcement learning algorithm known as Proximal Policy Optimization (PPO). In our approach, we use an Epsilon-Greedy Deep Q-Learning actor-critic model consisting of two separate deep neural nets in order to train the bot over a fixed time period. In order to train our agent, historical foreign exchange market data obtained from Forex software will be used [16]. The dataset acquired is in CSV format and consists of more than 50.000 instances of various currency pairs such as EUR/USD, USD/JPY, among others. Other libraries, such as mplfinance and OpenCV are used in order to adequately visualize the market's data [9, 13].

We have decided to pursue our research on this topic to gain a deeper understanding of the financial market and how autonomous trading agents trained with deep learning techniques can show significantly high performances. Moreover, we decided to make use of the PPO algorithm for our research given that this kind of actor-critic deep learning approach has not been used extensively for research in this domain [14]. Hence, the objective of this paper is to implement an actor-critic PPO deep learning model using Tensorflow's Keras in python.

1.1 Research Question

Currently, actor-only and critic-only approaches seem to be the most common for research of autonomous trading agents trained by deep reinforcement learning[26]. Moreover, the actor-critic PPO model chosen for this paper has yet to be researched extensively in the present literature for autonomous Forex trading agents, with only one example that compared the performance of PPO against other deep learning methods with a pre-defined trading strategy.[26]

This paper will focus solely on the PPO algorithm in order to build an autonomous Forex trading agent that can learn how to trade and maximize returns based on price predictions. The reason for choosing this model specifically is that it is relative to this research domain and has shown great results for autonomous agents in other fields. For example, the chosen algorithm for this paper has shown great performance when building autonomous agents in fields such as Football and robotics [20, 25]. Given the above-mentioned points and the current gap in the literature, the following research question will be studied and analyzed in this research paper:

How can deep reinforcement learning be used to build and train an autonomous Forex trading agent using the Proximal Policy Optimization (PPO) algorithm implemented with Tensorflow's Keras?

Sub-question: *Having analyzed the bot's performance, is it capable of generating profits after trading over a fixed period of time?*

2 Data Preparation

First of all, the visualisation of the data is introduced so the reader can have a clear understanding of the feature space and the instances of the dataset used. The plain market, without any technical indicators, consists of Candlesticks [12]; the name is given because they resemble candles: A typical candle, has a body, and a wick. However, in the case of a market candlestick, a wick on the bottom can be seen. Refer to Fig. 1 for illustration.

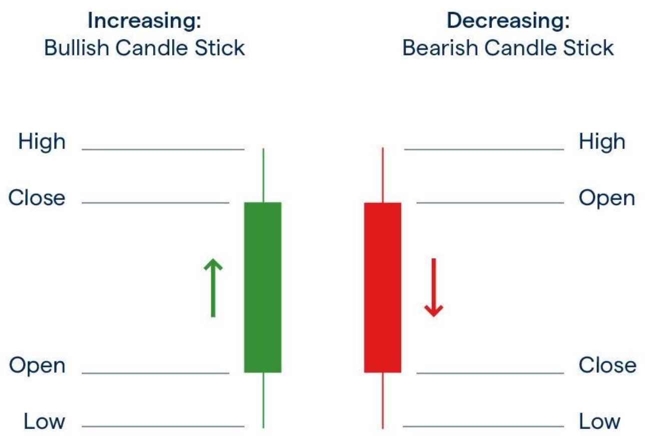


Figure 1: Market Candlestick[8].

The candlestick in Fig. 1, represents the chart known as OHCL (Open High Close Low) Data [24]. Therefore, the data consists of a data frame where each row corresponds to one candlestick in time. The data attached to each candlestick, is the price when the candlestick opened and closed, as well as when it was at its highest and lowest price points. As a result the dataset consists of four columns, each associated with each OHCL stage.

Aside from the columns mentioned above, the dataset consists of a fifth column called Volume. This represents the amount of people that invested in the market during that particular candlestick for a point in time.

To sum up, the CSV file represents hourly candlesticks and is sorted by date/time.

The dataset consists of approximately 50,000 instances for the following markets, EURUSD, USDJPY, GBPUSD and AUDCAD. Because of the importance of the timeline in this dataset, validation techniques such as cross-validation or using a validation set, were not implemented. This is because of time series data, in the case where cross-validation were to be used, then the model will be provided with the unfair advantage of looking at instances ahead of time, resulting in an inaccurate model.

Therefore when preparing the data, instead of using a validation set, the data was split 50-50: half of the data was used for training, and the other half for testing. Because of computational limitations, the data was split for shorter periods of time as an alternative for splitting the whole dataset. As an example: choosing a range of data for 1 month, means that the data now consists of 30 (days) x 24(hours) = 720 rows of instances.

It is important to note that the data is normalised inside the Custom environment, this is discussed further in `Section 3.2`. Moreover, since every currency pair has its unique exchange rate and every market has different Volumes (that is more people are trading the EURUSD market than the USDJPY market), normalising is crucial for Neural Networks. In other words, for the model to be fair and accurate, the input numbers are between 0 and 1. This also ensures that the model won't get stuck in flat regions. After normalising, the data preparation is done and the next step is to explain the method.

3 Method

3.1 Chosen Model

This section is regarding the reinforcement learning[21] model used for this report, created by OpenAI[19], known as the Proximal Policy Optimisation (or PPO) model[20]. PPO involves the collection of a small batch of experiences that are interacting with the environment, and then uses that batch to update its decision-making policy. In order to provide a higher-level understanding of this concept, the following example is provided.

Think of a policy like this: You're a tourist, and you're trying to get to the city centre. You have no clue where to go, so you ask a local, and they tell you to walk a certain path; that certain path - that you're going to follow - this is the policy.

The main intuition behind the algorithm is that after it updates its policy, the new policy shouldn't deviate too much from the policy before it. In order to achieve that, the PPO algorithm uses clipping to avoid large updates in its policy. By doing this, it leads to less variance, and introduces some bias; but ensures smooth training and makes sure that the model doesn't take senseless actions in the environment. The Neural Networks used in PPO are discussed in `section 3.1.2`.

3.1.1 Actor-Critic Model

The PPO model uses two separate Neural Network models, known as the Actor and the Critic. Firstly, based on the current policy that the PPO is using, the Actor Model performs the most important task of learning what action it must take. In case of the trading bot, the actor model would need to produce 3 actions: Buying, Selling, or Holding trades. Secondly, the Critic Model evaluates the environment, this is achieved by observing the reaction of the environment when the actor gives it an action. The critic then acquires a reward (which can be good or bad) based on the observation from the environment, and informs the Actor to adjust the policy accordingly.

If we go back to the tourist and the local example: The tourist's brain (the Actor) is telling his body (the environment) to walk straight, and the local (the critic) is watching him and telling him he's going the wrong way or the right way.

This results in the PPO to follow a loop of actions which can be seen in Fig 2[25]. Based on this figure, there are three parts that need to be implemented for a PPO Model. These include the Actor, the Critic and the environment. However, in order to visualise the results and evaluate how the model performed in the environment, a fourth part is also implemented called the Renderer. This is discussed in detail in `Section 3.2.4`. In short the Renderer generates GIFS for the moving environment.

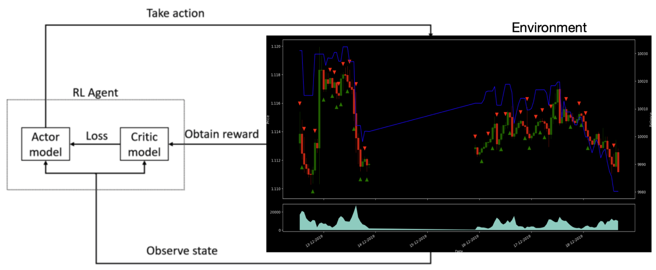


Figure 2: PPO Model Graph.

Note: that the PPO model implemented for the Trading Bot is a type of **Deep Q-Learning** model. This method of learning is widely used for the purpose of trading[27]. Deep Q-Learning is a model that offers a discounted reward system: This is the calculation of all future rewards (discounted) plus the current reward to determine the current action it must take;

However, in the case of the Trading Bot, a softmax activation on the outputs is used, to get a probability distribution - this is usually just a **tabular Q-learning** approach[7], but we use it in combination with other **Deep Q-learning** methods. The probability distribution is then fed into an Epsilon-Greedy method[3] to determine which action the model will take.

The reason why the model is still considered a **Deep Q-Learning** model, is that reward is not calculated after each episode, but rather after each step. In other words, the environment is run by an Epsilon-Greedy algorithm following the current policy. After each step in the environment, a reward is observed, and the target value is computed. This value consists of the observed reward + the discounted future rewards in the new states: the entropy of our output. The difference between this entropy and the observed loss is calculated to get the total loss. Back-propagation can then be applied. This is the intuition behind the algorithm that was used for the bot. See `Section 3.4` for a higher-level understanding of the implemented PPO model.

3.2 Custom Environment

This section covers the custom environment that was created for the FX Trading Bot. In order to implement this environment a class was created which consisted of a number of variables and functions. This class achieved the simulation of a moving environment. In order to give a better understanding on how the environment was implemented, the most important variables are listed below:

Starting balance, net worth, data-frame (containing all the data), amount of steps it would need to take, a list for the action space (all the possible actions the model can take), a list containing the order history (what orders it took, and where), and a few other minor variables.

To achieve this simulation of a moving environment, four different functions were implemented. These are discussed in detail in the sections below so the reader can have a deeper understanding of their functionality.

3.2.1 Reset

The Reset function in short, resets the environment back to the starting stage so it can be ready for the next episode. When the model receives data, in this case this would be FX data for a period of about a month, it starts training. As it trains, it steps through every candlestick and makes an action, it then calculates the reward for each candlestick and eventually updates the policy. The end of the episode is reached once the model steps through all the candlesticks in the given dataset.

When training in reinforcement learning, this is typically regarded asonline training, therefore the model is trained a number of times. Each training is considered a single episode; the number of episodes needs to be specified beforehand.

When the end of an episode is reached, the model follows a certain policy. This policy tends to be bad in the first episodes and develops to a more suitable one as the model continues to train. Every time the model starts a new episode, it trains using the same FX market dataset. Therefore, for this to be successful, the market needs to be reset so the model can be placed back at the same starting point.

Think of the tourist and the local example: The tourist eventually makes his way to the city centre, but it was poorly done. But now the tourist knows his way to the city centre (even though its performance is poor). So to better his performance, he goes back to the starting position and walks again to the city centre, this time doing it better and therefore have a better performance.

Thus the reset method needs to be used to reset the environment and build up the market so it's ready for the next episode.

3.2.2 Step

The Step function is a key for the implementation of the bot. The stepping function allows the market to progress to the next candlestick. In the case of the Trading Bot, aside from moving to the next state, the Step function also takes as input an action and performs that action to the following step.

A pseudo-explanation of how the function works is given here. Firstly, one of three actions are passed in: 0, 1, 2 corresponding to Buy, Sell, Hold respectively. If the action passed in is BUY, the model will enter the market with 1% of the current balance on the current step, but will only do so if there isn't an existing order. One thing to note is that the FX Trading Bot model can only be in order at a time. In the case where the action passed is SELL, the model will only sell 100% of the order, if there is a current order being held in this current state. Lastly, if the action is HOLD, the model simply does nothing.

After the action is passed in, it then sets the new Net worth of the environment, appends all the information to the order history, and calculates the reward. The reward is a very significant part of the model. For the bot, a simple naive reward was used, where it is positive if the bot made money, and negative if the bot lost money between the current and the previous state. The reasoning behind the naive reward is discussed in the results section (Section 5).

3.2.3 Next Observation

For the discounted reward system to work, the model needs to take into account the future states. Therefore, the Next Observation function simply returns the state at the next step. This function is used at the Step function since the next state needs to be returned. This needs to be done so the Step function can perform calculations for the discounted reward system.

3.2.4 Render

In order to visualise the environment, an image of each step needs to be captured, resulting in a GIF generation. The Render function gathers all the information necessary at the current step, and then passes it into another function that creates a plot, and renders it as an image. The visualisation will be discussed in `Section 3.3`.

3.2.5 Get GAEs

This function houses all the calculations for the discounted reward system for the Q-Learning algorithm. To start off, a simple definition of Advantages. Advantages are how much better off the model is by taking a particular action in a specific state, they are calculated using the collected rewards from each state. For example, in the case of a good action - making an order that leads to a profit - the advantage of taking that action needs to be calculated.

To perform this calculation, the Generalised Advantage Estimation (GAE) is used. To best explain GAEs, a reference to the Monte-Carlo return algorithm[10] needs to be made. The algorithm is referred as follows:

$$R_t = \sum_{l=0}^{T-t} \gamma^l r_{t+l}$$

where R is the unbiased sample of the expected return at a given state t.

However, each reward R_t can be a random variable due to stochasticity from the change of the environment and policy. To accompany this, an n step return can be created that can provide a lower variance estimator, but this again returns to the bias-variance trade-off, thus resulting in a higher bias. The n step return can be computed by truncating the sum of returns after n steps, and approximating the returns from the remaining steps using the value function. In other words, this is the discounted reward system that was discussed earlier. The resulting n step return can be written as

$$R_t^{(n)} = \sum_{l=0}^{n-1} \gamma^l r_{t+l} + \gamma^n V(s_t + n)$$

where $R_t^{(1)}$ results in the 1-step return used in Q-Learning and R_t^{∞} results in the original Monte-Carlo return. While R_t^{∞} provides an unbiased, high-variance estimator; $R_t^{(1)}$ provides a biased, but low-variance estimator. In order to fine tune this trade-off, λ-return[18] estimator can be calculated as an exponentially weighted average of n steps return using a parameter λ as the decay coefficient. This can be written as:

$$R_t(\lambda) = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^n$$

Assuming that all the rewards after step T are 0 (where $R_t^n = R_t^{T-t}$ for all $n \geq T-t$), the infinite sum can be calculated using the following equation:

$$R_t(\lambda) = (1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^n + \lambda^{T-t-1} R_t^{T-t}$$

where $\lambda = 0$ yields the 1-step return $R_t^{(1)}$; and $\lambda = 1$ yields the Monte-Carlo return R_t^{∞}.

Values in between 0 and 1 will yield different forms of the formula that can be used to balance the bias and variance of the value estimator. In Q-learning, the value function is updated by adding the reward to the value function calculated for following state, this can be seen in pseudocode in the `Appendix`[4]. Regarding the trading bot, the advantage is estimated using the λ-return, which yields the Generalised Advantage Estimator $GAE(\lambda)$

$$\hat{\mathbb{A}} = R_t(\lambda) - V(s_t)$$

Formula A: Policy updates are performed using a batch of experiences collected (R = rewards, A = dones, S = values, N = nextValues). In Fig. 3 the pseudocode is shown:

```
def getGAEs(R, A, S, N):
    δ <- [r + γ*(1-a)*n - v for r,a,n,v in R,A,N,S]
    for i in reversed(δ):
      δ[i] <- δ[i] + (1-A[i])*γ*λ*δ[i+1]
    target <- δ + S
    δ <- (δ - δ.mean) / δ.std
    return δ, target
```

Figure 3: getGAEs Pseudocode.

Note: The value $a =$ dones was passed as a supplementary part of the GAE formula. This was added because of the custom environment, so when the bot stops trading the last received rewards needs to be used as a discount.

γ is a constant known as the discount factor. It is used in order to reduce the value of the future state to emphasise the current sate. In other words, the current state is more valuable than any future state. This value is set to 0.99 for the model. λ is a smoothing parameter, as explained above, for reducing the bias at the cost of some variance. This gives the advantage of taking an action both in the short term and in the long term. This value is set to 0.95.

In the last step, the result is normalised by subtracting the mean and dividing by the standard deviation.

3.2.6 Replay

This function is in charge of fitting the model with its required data; all the states, actions, predictions and next states are collected. Then the predict function gets called with two parameters (the states and next states) to update them.

The target and the advantages are calculated by calling the getGAEs function; and finally, the required data fits the Actor model and the Critic model. The model's Fit function trains the model, but only for the current step.

3.2.7 Act

The final function we discuss is the Act function. This is considered a helper function since it returns an action and a prediction by calling the Actor Model predict function. This in turn returns a probability distribution, which in turn is passed into *Numpy's Random Choice*[6] function; this chooses a random action based on the probability of the predictions. This is known as the Epsilon-Greedy algorithm that was explained in `Section 3.2.1`

3.3 Visualisation

The visualisation techniques used for the environment are explored here. To visualise OHLC data, a rendering technique was implemented, where a plot is drawn at each step, and an image of that plot is then appended onto a GIF for later analysing.

These rendering techniques take a lot of computational power and as a result, this technique is only done when Testing the model. This is because if it were used to visualise the trained model, it would take a tremendous amount of time, and hence generate a large file.

Two Python packages were used to achieve the visualisation: **mplfinance**[9] and **OpenCV**[13].

3.3.1 Mpl Finance

Firstly, the mplfinance package works using Pandas[23] and Matplotlib[15] to generate OHCL plots. In short, it takes in the OHCL data and generates a plot.

```
mpf.plot(daily,type='candle')
```

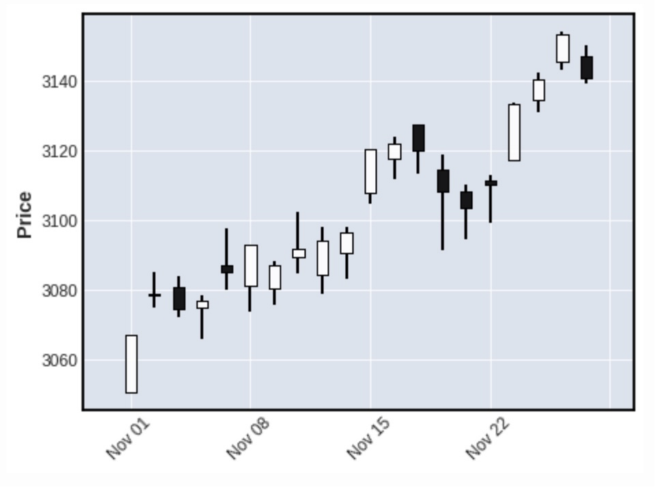


Figure 4: Example Code for Candlestick Plotting.

The package has multiple parameters that allow for different colouring. A selection of green for increasing candles (Bullish) and red for decreasing candles (bearish) reference Fig. 1 was made for the trading bot.

It's important to note that this plot was generated purely using Matplotlib; this means properties such as subplots and twin axes can be used.

Beside from the plot in Fig. 4, a subplot was created underneath that shares the x-axis, which is used to plot a filled line graph of the Volume. The y-axis was shared to generate a line graph representing the Net Worth. This was done to clearly visualise when the bot made a profit or loss.

3.3.2 Open CV

Secondly, the OpenCV package provides real-time optimised Computer Vision tools; this is used to generate a moving plot. As a line of data is added to a text file, OpenCV updates the plot in real-time. It also has the functionality of working with Matplotlib; allowing to draw shapes on the plot.

This added functionality is used to draw red and green arrows to represent when the model entered the market.

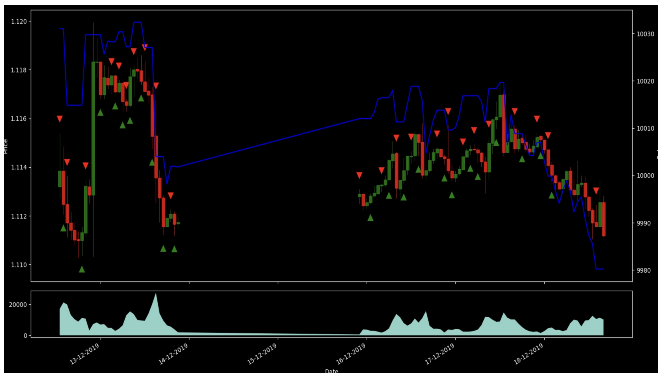


Figure 5: Environment Example.

Fig. 5 is an example taken from one of the frames in a generated GIF. This shows how the market is rendered once the model is tested; since this is a GIF the picture moves through the specified amount of time but can't be shown in this report.

3.4 PPO Model

3.4.1 Actor Model

To start with, the actor model was created using a Neural Network that takes in, as an input, a state size and flattens it. This state size is the amount of candlesticks that the model is allowed to look back on. This input is then passed through a three Dense layer - first with 512 nodes, followed by 256 nodes and then 64 nodes - all using ReLU activation[1]. As an output, the model passes through one more Dense layer, taking in the Action Space and having a *softmax* activation[28] for a probability distribution.

After the above steps are complete, the model is then compiled using the Adam optimiser[5] and a **custom PPO loss**.

Note: that Tensorflow's Keras[22] was used for all the model implementation.

3.4.2 Custom (Actor) PPO Loss

This is where the policy of the PPO model is implemented. This is done by referring to the following formula[20]

$$L_{CLIP} = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

where epsilon is a hyper parameter $\epsilon = 0.2$; this is the most optimal value for epsilon[20]. To have a more clear understanding on how this formula works for the model, it must be noted that the probabilities and old-probabilities represent the policy of the model. Hence the aim is to improve these probabilities and in return receive better actions over time.

A challenge that this model has faced when it comes to reinforcement learning, is in the case of adopting a bad policy. When that is the case, the model only follows bad actions in the environment, resulting in an unrecoverable path in training. PPO addresses this problem by making small updates to the policy and hence, disallowing to make major changes; this is achieved using the *clipping* parameter. Furthermore, $\hat{A}_t$ in the formula represents the advantages for a given state t; for positive advantages, the formula takes $1+\epsilon$ and takes $1-\epsilon$ for negative advantages. It then clips the probability ratio which removes the incentive for moving r_t outside the interval $[1-\epsilon, 1+\epsilon]$; the minimum is taken between the clipped and unclipped objective, making the final objective a lower-bound. This is visualised to give a better understanding in Fig. 6 [20]

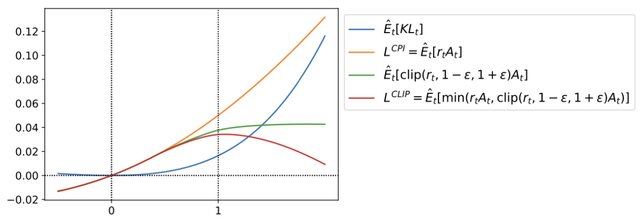


Figure 6: Linear Interpolation Factor.

The red line is the lower bound of the green line, which is the desired result. This avoids big updates in the policy.

In practice, the probability and the old-probability are extracted and clipped, resulting in the upper and lower-bound. The negative mean of the element-wise minimum of the clipped probabilities is then returned; this results in the actor loss.

3.4.3 Critic Model

The critic model is implemented in the same manner as the Actor Model, aside for two differences: No activation function is used to compile the output, and the Mean Squared Error (MSE) loss is used. This change in implementation is because the environment just needs to be evaluated on the performance of the Actor Model.

4 Training

Now that all the necessary functions are established, the training of the model can commence. The algorithm in Fig.7 was used to train the model.

```
def train(environment, batchSize, episodes):
  loop through episodes:
    state <- environment.reset()
    loop through batchSize:
      environment.render(False)
      action, prediction <- environment.act(state)
      nextState, reward, done <- environment.step(action)
    environment.replay()
```

Figure 7: Training Pseudocode.

Note: that the replay function is in charge of doing the training.

In the first round of training, 350 episodes were trained, with a batch size of 500, on 720 steps of data (720 steps = 24 hours × 30 days) with a learning rate of 1×10^{-5}. Following the first training, this time the model was trained for 3000+ episodes, with a batch size of 500, on 2160 steps of data (2160 steps = 24 hours $\times$ 30 days $\times$ 3 months) with a learning rate of 3×10^{-4}.[20]

5 Results

Concerning the first training round, it was evident that the actor model learnt a specific policy and insisted to follow that policy. This occurrence can be visualised in the Tensorboard in Fig. 8.

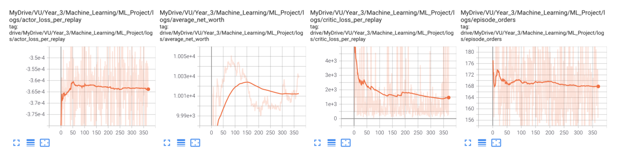


Figure 8: Tensorboard Results for First Training.

The far left plot is the Actor Loss (PPO loss), followed by a plot of the Net Worth, the Critic Loss (MSE) and lastly, the amount of orders the model makes per episode. The performance for each function gives rise to the expected results: The Actor Loss is meant to rise due to the nature of the model trying to better its policy. The Critic Loss also follows the expected path, as well as the Net Worth - however, it peaks around 2.1% profit, and averages around 1.25% profit after 1 month of data. An unexpected result was the episode orders: which fluctuate around 168 orders per episode; that implies that around 23% of the steps taken were market entries - this value is high considering that the time period is only one month. The number of episodes and the learning rate are therefore adjusted and increased because of the above results.

The new parameters are changed and the model is trained again. This leads to the following results:

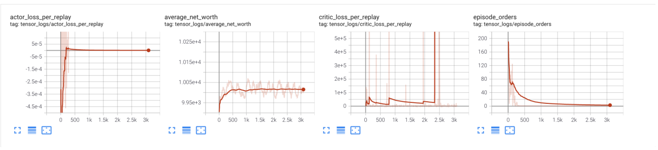


Figure 9: Tensorboard Results for Latter Training.

Based on the results shown in Fig. 9 the slow learning rate is performing better. However this ends up in altering behaviour of the rest of the graphs in an unexpected way. What stands out the most is the Actor and Critic Losses. In particular, the Actor Loss converges in a very rapid manner to a specific policy and remains in the same spot after 250 episodes. This event can also be seen in the episode order graph on the right where the model converged to only taking a single order per episode.

To explain why this occurs, the naive reward system needs to be recalled. The model receives a positive reward if the current network is higher than the previous net worth, and a negative reward for vice versa.

However since the model is never punished for taking a completely different task, the model realises that the best action to take is to enter the market and hold on to the one trade as long as it can. If that one trade it holds on to is a profit, then the current net worth will always be higher than the previous, hence the model receives a positive reward even if it just entered one trade.

That is essentially the policy it stuck to: Enter the market and hold. Thus the market is not optimising for max profits; this is not necessarily a bad thing, but it's not the goal of this research report. Another thing to mention is that the model was trained on the USDJPY market over 3 months worth of data; those 3 months were always a random set of continuous 3 months (e.g. each episode could result in 3 months of data between 2013 and 2021).

Therefore, when the model was tested on unseen USDJPY data, it always performed well: making a profit between 2.1% and 2.8% a month.

In Fig 10. The results of testing on 10 different sets of 1-month USDJPY data can be seen; which averaged around 2.29% Profit. Therefore the model performs well on the market it was trained on.

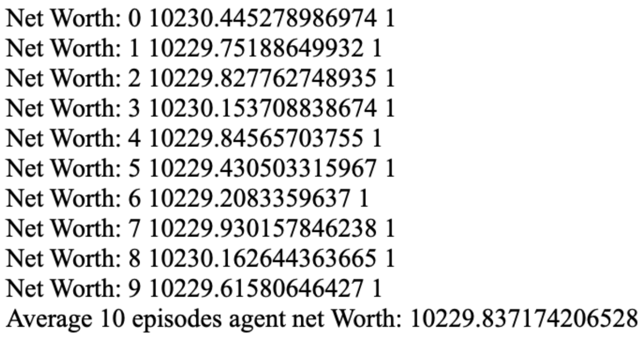

```
Net Worth: 0 10230.445278986974 1
Net Worth: 1 10229.75188649932 1
Net Worth: 2 10229.827762748935 1
Net Worth: 3 10230.153708838674 1
Net Worth: 4 10229.84565703755 1
Net Worth: 5 10229.430503315967 1
Net Worth: 6 10229.2083359637 1
Net Worth: 7 10229.930157846238 1
Net Worth: 8 10230.162644363665 1
Net Worth: 9 10229.61580646427 1
Average 10 episodes agent net Worth: 10229.837174206528
```

Figure 10: Test Results on USDJPY.

On the other hand, when testing the model on EURUSD data using the same model, we get the following results shown in Fig. 11. In this case, a -0.82% profit was made on an average of 10 different sets of 1-month EURUSD data.

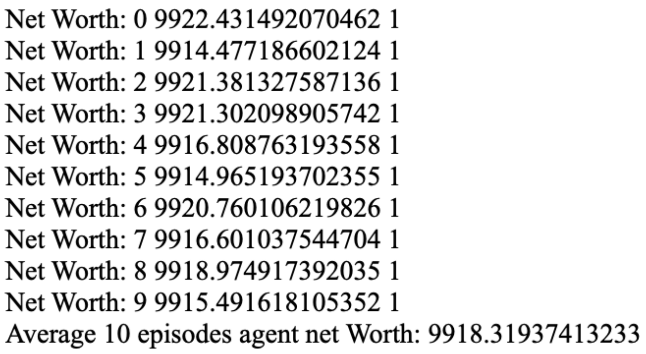

```
Net Worth: 0 9922.431492070462 1
Net Worth: 1 9914.477186602124 1
Net Worth: 2 9921.381327587136 1
Net Worth: 3 9921.302098905742 1
Net Worth: 4 9916.808763193558 1
Net Worth: 5 9914.965193702355 1
Net Worth: 6 9920.760106219826 1
Net Worth: 7 9916.601037544704 1
Net Worth: 8 9918.974917392035 1
Net Worth: 9 9915.491618105352 1
Average 10 episodes agent net Worth: 9918.31937413233
```

Figure 11: Test Results on EURUSD.

Note: There is a number 1 after each episode; this is just the amount of orders the model makes during that episode, which we have established has converged to 1.

6 Conclusion

In conclusion, the rise in computational power and machine learning algorithms has paved the way for research of automated trading agents in the market. These used to favour heavily towards support vector machines or neural networks, however this trend has shifted towards reinforcement learning algorithms given its capacity to trade autonomously and learn market behaviour. The goal of this paper was to apply reinforcement learning within forex trading, using the PPO algorithm, as its performance has yet to be extensively researched. The research question stated: *How can deep reinforcement learning be used to build and train an autonomous Forex trading agent using the Proximal Policy Optimization (PPO) algorithm implemented with Tensorflow's Keras?*

The data acquired included features on the Open, High, Close, Low and Volume, with 50,000 instances of various forex markets which were normalized. Due to the nature of time series data, validation techniques such as cross validation are incompatible. Fortunately, a 50-50 split between training and test data gives sufficient examples to test on. However, shorter periods of time are used due to computational limits.

The PPO model chosen for this research is an actor-critic model, which uses a Deep Q-learning discounted reward system to determine the possible actions. To select the most optimal, softmax activation and Epsilon Greedy policy is used. The observed reward and discounted future rewards is the entropy of the output. This in turn is used with observed loss to give total loss which can be minimized by backpropagation. A custom environment was implemented to simulate and visualize a moving environment.

Training was conducted in two rounds. The first round consisted of 350 episodes on 720 steps (1 month) of data, while the second of 3000+ episodes on 2160 steps (3 months). During which, the batch size remained at 500 for both and the learning rate was increased from 1×10^{-5} to 3×10^{-4}. The performance of the second model outdid the first, but had derived very quickly to a policy of entering and holding due to the naive reward system. The model averaged a profit of 2.1% to 2.8% per month on unseen data for USDJPY and -0.82% on unseen EURUSD. In regards to the research question, these models may be perceived as a success. However, further improvements may be realised but could not due to time limitations.

6.1 Further Research

There are a number of implementations that can be done to better this model. For starters, optimising the reward strategy. A reward system that could better the current naive implementation, could be to gradually punish the Actor for holding onto trades; `ie.` the longer the model holds onto a trade, the more you punish it. This, in turn, could incentivise the model to make more short term trades, as this is what would optimise for profitability. In other words, punish the model for doing nothing.

Further Deep learning model techniques could also be implemented to better the model. For instance, in trading, it is known that history repeats itself; to that extent, it may be a good idea to have a memory recall. A Long-Short Term Memory (LSTM)[11] model could be used to achieve that. Trading is also all about patterns in the market, and recognising those patterns could lead to substantial profits; therefore, a Convolutional Neural Network (CNN)[17] model can be used to better the agent, since patterns are always occurring in the present time.

Additionally, technical indicators can be added into the model as extra data. This might improve the predictions of the model. On the other hand, it would be necessary to punish the model for relying too much on these technical indicators, as they lead to many false-positives in the market. A discounted system could be used in this case, like the one used for the rewards - so that the model wouldn't focus too much on the indicators, but rather use it as an indication that it's correct.

It could also be worth while finding a way to train the model over all 12 years worth of data, for all markets. This would need an substantial load of computational power and resources; it would possibly maximise the model's predictability.

Finally, further research could be done about automation and deployment of this model in the real world. For example, integrating the model using an API to trade using a broker, as well as having techniques to automatically download the latest data for the model. Although, for the people to actually choose to deploy a model, it would be at their own risk since the current model could make a loss in the long run.

References

[1] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[2] Akhil Raj Azhikodan, Anvitha G. K. Bhat, and Mamatha V. Jadhav. Stock trading bot using deep reinforcement learning. In H. S. Saini, Rishi Sayal, A. Govardhan, and Rajkumar Buyya, editors, *Innovations in Computer Science and Engineering*, pages 41–49, Singapore, 2019. Springer Singapore. URL: `https://doi.org/10.1007/978-981-10-8201-6_5`.

[3] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[4] Peter Bloem. Reinforcement learning lecture. *X_400154 Vrije Universiteit Amsterdam*, page 50, 2021.

[5] Jason Brownlee. Gentle introduction to the adam optimization algorithm for deep learning, Jan 2021. URL: `https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/`.

[6] The SciPy community. numpy.random.choice¶. URL: `https://numpy.org/doc/stable/reference/random/generated/numpy.random.choice.html`.

[7] Marco Dorigo and Hugues Bersini. A comparison of q-learning and classifier systems. *From animals to animats*, 3:248–255, 1994. URL: `https://dl.acm.org/doi/10.5555/189829.189896`.

[8] Patrick Foot. Japanese candlestick trading guide. URL: `https://www.ig.com/za/trading-strategies/japanese-candlestick-trading-guide-200615`.

[9] Daniel Goldfarb. mplfinance. URL: `https://pypi.org/project/mplfinance/`.

[10] Stefan Graf and Ralf Korn. A guide to monte carlo simulation concepts for assessment of risk-return profiles for regulatory purposes. *European Actuarial Journal*, 10:273–293, 2020. URL: `https://doi.org/10.1007/s13385-020-00232-3`.

[11] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. LSTM: A search space odyssey. *CoRR*, abs/1503.04069, 2015. URL: `http://arxiv.org/abs/1503.04069`, `arXiv:1503.04069`.

[12] Adam Hayes. Candlestick, Aug 2020. URL: `https://www.investopedia.com/terms/c/candlestick.asp`.

[13] Olli-Pekka Heinisuo. opencv-python, 2021. URL: `https://pypi.org/project/opencv-python/`.

[14] Bruno Miranda Henrique, Vinicius Amorim Sobreiro, and Herbert Kimura. Literature review: Machine learning techniques applied to financial market prediction. *Expert Systems with Applications*, 124:226–251, 2019. URL: `https://www.sciencedirect.com/science/article/pii/S095741741930017X`, `doi:https://doi.org/10.1016/j.eswa.2019.01.012`.

[15] John D. Hunter and Michael Droettboom. matplotlib. URL: `https://pypi.org/project/matplotlib/`.

[16] Forex Software Ltd. Download historical forex data. URL: `https://forexsb.com/historical-forex-data`.

[17] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[18] Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *CoRR*, abs/1804.02717, 2018. URL: `http://arxiv.org/abs/1804.02717`, `arXiv:1804.02717`.

[19] John Schulman. Proximal policy optimization, Sep 2020. URL: `https://openai.com/blog/openai-baselines-ppo/`.

[20] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL: `http://arxiv.org/abs/1707.06347`, `arXiv:1707.06347`.

[21] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. *MIT press*, 2018. URL: `https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf`.

[22] Keras Team. Keras documentation: About keras. URL: `https://keras.io/about/`.

[23] The PyData Development Team. pandas. URL: `https://pypi.org/project/pandas/`.

[24] Michael C Thomsett. *Basic Candlestick Chart Investing.* Pearson Education, 2010.

[25] Chintan Trivedi. Proximal policy optimization tutorial (part 1: Actor-critic method), Jun 2020. URL: `https://towardsdatascience.com/proximal-policy-optimization-tutorial-part-1-actor-critic-method-d53f9afffbf6`.

[26] Chun-Chieh Wang and Yun-Cheng Tsai. Deep reinforcement learning for foreign exchange trading. *CoRR*, abs/1908.08036, 2019. URL: `http://arxiv.org/abs/1908.08036`, `arXiv:1908.08036`.

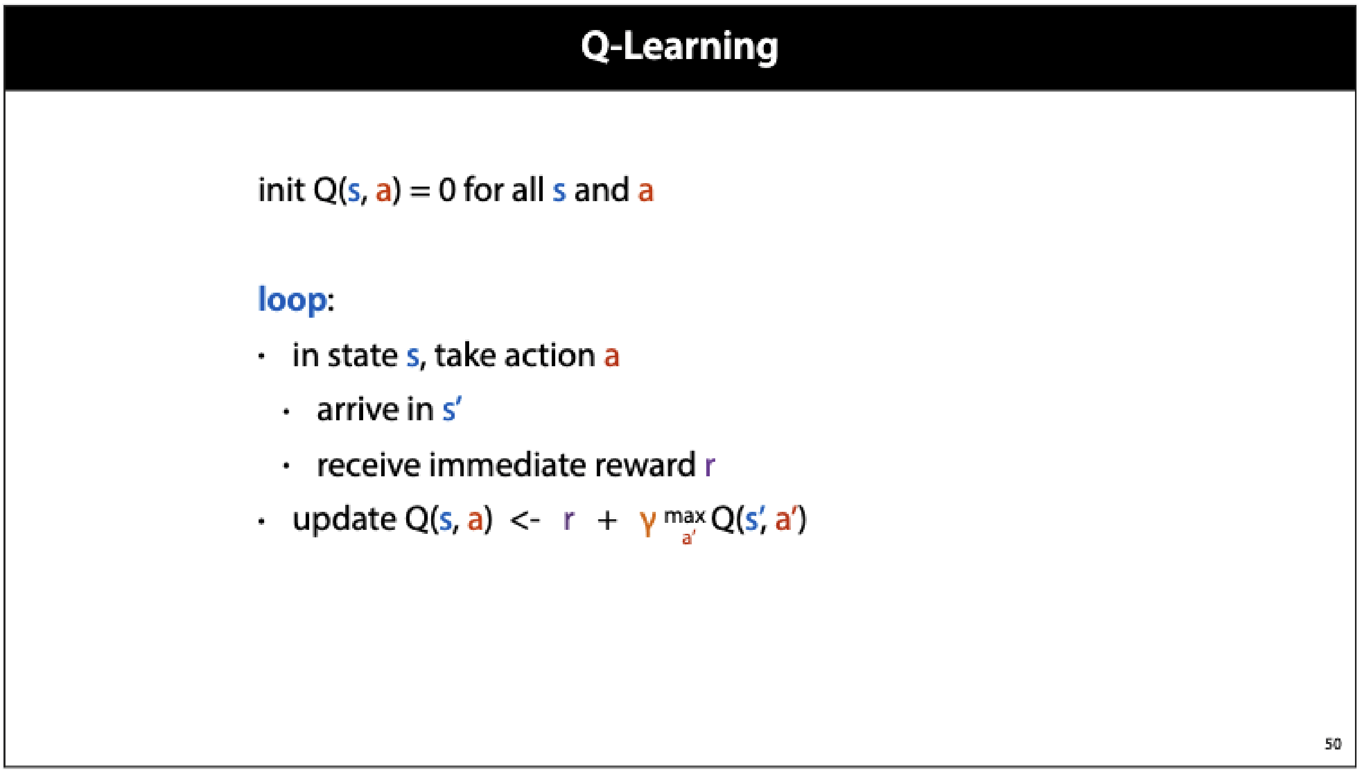
[27] Yang Wang, Dong Wang, Shiyue Zhang, Yang Feng, Shiyao Li, and Qiang Zhou. Deep q-trading. *cslt. riit. tsinghua. edu. cn*, 2017. URL: `http://cslt.riit.tsinghua.edu.cn/mediawiki/images/5/5f/Dtq.pdf`.

[28] Thomas Wood. Softmax function, May 2019. URL: `https://deepai.org/machine-learning-glossary-and-terms/softmax-layer`.

[29] Zhuoran Xiong, Xiao-Yang Liu, Shan Zhong, Hongyang Yang, and Anwar Walid. Practical deep reinforcement learning approach for stock trading. *CoRR*, abs/1811.07522, 2018. URL: `http://arxiv.org/abs/1811.07522`, `arXiv:1811.07522`.

[30] Zihao Zhang, Stefan Zohren, and Stephen Roberts. Deep reinforcement learning for trading. *The Journal of Financial Data Science*, 2(2):25–40, 2020. URL: `https://jfds.pm-research.com/content/2/2/25`, `arXiv:https://jfds.pm-research.com/content/2/2/25.full.pdf`, `doi:10.3905/jfds.2020.1.030`.

A Appendix



Q-Learning

init Q(s, a) = 0 for all s and a

loop:

- in state s, take action a
  - arrive in s′
  - receive immediate reward r
- update Q(s, a) <- r + $\gamma \max_{a'} Q(s', a')$

50

Figure A.1: Slide 50 - Reinforcement Learning Lecture - ML Course 2021 - VU[4].

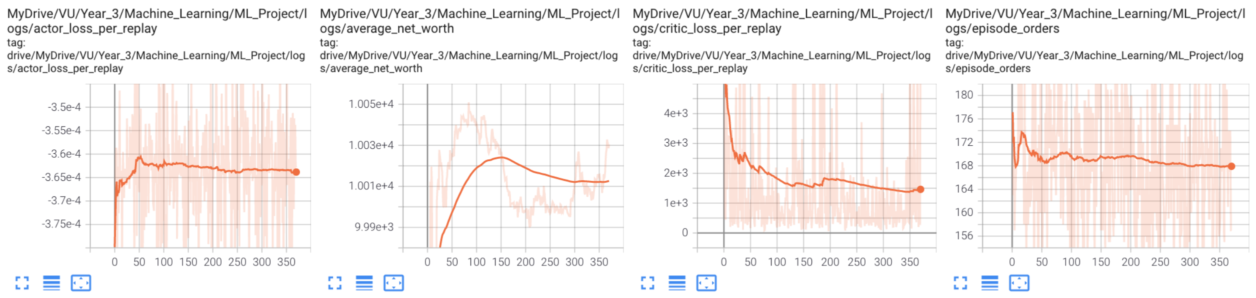


Figure A.2: Tensorboard Results for First Training.

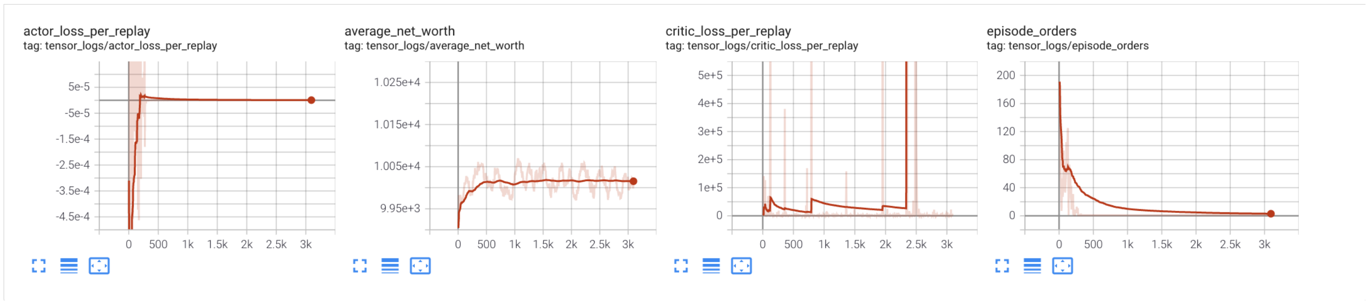


Figure A.3: Tensorboard Results for Latter Training.