Automated Reinforcement Learning Trading Agent

Author: Roberto Lentini
Supervisor: Rafal Kustra

CHAPTER 1: PROJECT INTRODUCTION 2

SECTION 1.1 INTRODUCTION 1
Subsection 1.1.a Reinforcement Learning 1
Subsection 1.1.b Reinforcement Learning in Finance 5
SECTION 1.2 ALGORITHMS 6
Subsection 1.2.a Categories of Reinforcement Learning 6
Value Based Algorithms 6
Deep Q-Learning (DQN) 8
Double DQN (DDQN) 8
Dueling DQN 8
Gated DQN (GDQN) 8
Rainbow DQN (RDQN) 8
Double DQN (DDQN) 8
Value Based Algorithms for Automated Trading 9
Policy Gradient Algorithms 10
Advantage Actor Critic (A2C) 10
Gated Policy Gradient (GDPG) 11
Proximal Policy Optimization (PPO) 11
Policy Gradient Algorithms for Automated Trading 12
SECTION 1.3 ABOUT THE DATA 12

CHAPTER 2: PROJECT RESULTS 13

SECTION 2.1 FINAL ALGORITHM, TRADING RESULTS AND GRAPHS 13
SECTION 2.2 CHALLENGES AND FUTURE STEPS 17

CHAPTER 3: PYTHON CODE DOCUMENTATION 17

SECTION 3.1 MODIFYING THE CODE 17
SECTION 3.2 ENV.PY 20
SECTION 3.3 MODELS.PY 21
SECTION 3.4 UTILS.PY 22
SECTION 3.5 MAIN.PY 26
SECTION 3.6 CUSTOMIZING ENVIRONMENT DATA 27

CHAPTER 4: TRADING DASHBOARD USER INTERFACE 27

SECTION 4.1 GETTING STARTED WITH THE DASHBOARD 27
SECTION 4.2 USER INTERFACE USAGE GUIDE 29
SECTION 4.3 MODIFYING TRADING STRATEGY 30

Chapter 1: Project Introduction

Section 1.1: Introduction

This project will consist of looking at different reinforcement learning algorithms and then apply the shared proximal policy optimization algorithm to Ethereum data. The algorithm will make decision on when to buy and trade the currency. The goal will be to beat the "buy-and-hold" method most passive investors use.

Reinforcement Learning

Machine learning is a sub field of artificial intelligence which uses past data to be able to create new models that are capable of classifying, predicting, or acting. The architectures of a standard machine learning model, such as a supervised model, will work in a linear fashion as a static function. If we use a supervised learning model to solve an image classification problem, we will be passing data into the model to get the output of the image category. It is essentially passing data to a model to get an output.

Reinforcement learning is a sub field of machine learning that seeks to solve a different kind of problem that standard machine learning cannot solve efficiently. It tackles control tasks. A control task is a scenario where an algorithm needs to learn how to act and not just classify or predict like in standard machine learning. All the data in a control task lives in time and space. The algorithm's future decisions will be influenced by its past decisions. This means that a reinforcement learning algorithm framework is built in a loop format as opposed to linear format.

Let us evaluate how both types of algorithms would handle a control task to understand the difference between machine learning and reinforcement learning. To be able to make a self-driving car we would need to teach the car how to drive properly in a simulation. If we were to tackle this issue using a supervised learning method, we would need people to label the data frame by frame. This would allow the algorithm to learn the correct decisions for each scenario the car would face. This would accumulate to endless hours of labeling, and it would be virtually impossible to label each scenario the car would face. Instead, if we use reinforcement learning, we do not need to tell the car what the right thing to do at every timepoint is. The car's "brain" would learn how to drive and act in different scenarios through repetition, thus, learning how to act in future scenarios.

The essence of reinforcement learning can be described as such: we have the learning algorithm which takes an action while attempting a control task; i.e., driving a car. The learning algorithm will get a reward which will positively or negatively reinforce the action that it took. This is much like trying to teach a dog a new trick. A person will give the dog a treat when it successfully completes the trick which will help reinforce the trick in the dog's brain.

Understanding the basics of the reinforcement learning architecture requires knowing a handful of terms: state, action, reward, environment, and agent. The agent is the part of a reinforcement learning algorithm that processes inputs to decide what action to take. In our self-driving car example, the car's "brain" would be what we refer to as the agent. The environment is the world which the agent lives in.

Essentially, the environment is what generates the input data for our agent. In the self-driving car example, the environment refers to the simulated world the car (our agent) is situated in. The state is information from the environment at a certain timestamp

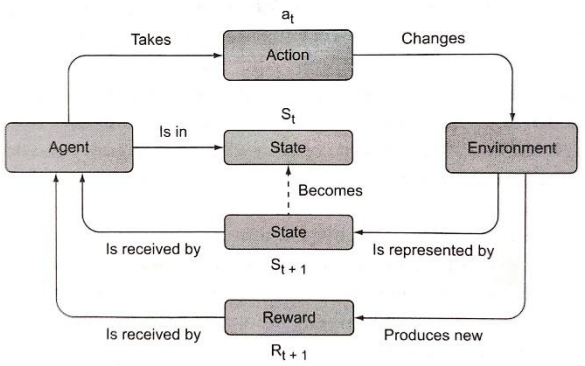

Figure 1.0 A diagram of a reinforcement learning framework from [1] p.25. An agent takes an action in its current state. This action will modify the environment and bring a new state to the action. The action will also cause a reward which the agent will use for its next action decision. These rewards are used to reinforce the agents learning.

which the agent uses to decide how to act. In the self-driving car example, the state could just be the open road at one point or, if the car encounters a stop sign, then the state would be updated, and the agent will need to decide what to do based on this input from the environment. If we think of a game of chess, the state of the board would be the position of all the chess pieces at a certain time stamp. An action is the decision taken by the agent which will change its environment and lead to a new state. If our self-driving car sees a stop sign and stops, then it has taken the action to stop. A reward is a positive or negative signal that is given to the agent after it has taken an action. If our self-driving car took the action to stop when it evaluated the current state, seeing a stop sign, then it would receive a higher reward as opposed to not having stopped (under the assumption that we are teaching the agent how to drive obeying to traffic laws). The agent's goal is to maximize its reward, therefore, its good actions will be reinforced using a higher reward.

We can now build from the general architecture of reinforcement learning and look at the theoretical processes that allow it to work. The theory behind reinforcement learning builds on the Markov process, also known as the Markov chain. The Markov process is a system which describes a sequence of possible events and states. This sequence of all possible states in an environment is known as the state space. The states in the system need to fulfill the Markov property, which means that the system does not have a memory. A future state can only depend on the current system's state. The Markov Decision Process (MDP) is an extension to the Markov process where the system now takes into account an action space, set of all possible actions, and rewards. Let us look at an example of a Markov Decision Process. Let us say we have a baby with the following state space: crying, sleeping and smiling. The action space will consist of feed and do not feed. We will also attach rewards to the states. If the baby is smiling the reward will be +2, if the baby is sleeping the reward will be +1 and if the baby is crying the reward will be -1. The entire system can be seen in figure 2.0.

If we have a crying baby and decide not to feed it, then there's a 0.95 probability that the baby will remain in the current state of crying giving the system a reward of -1. If

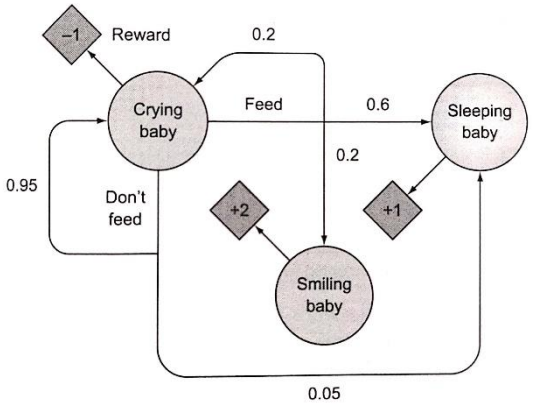

Figure 2.0 A diagram of the Markov Decision Process (MDP) [1] p.49.

we feed a crying baby, then there is a probability of 0.6 that the baby will transition to a state of sleeping giving the system a reward of +1. The rest of the system follows the same pattern. The goal of our agent in a reinforcement learning problem is to maximize the reward of the Markov Decision Process and this is the way the agent learns how to act in a control task.

The reinforcement learning agent uses value functions and a policy function to be able to maximize its reward in a Markov Decision Process. The policy function is mapping from states to actions given those states. The value functions refer to the expected reward of being in some state or taking some action. We have two value functions: the state-value, which maps state to the expected reward, and the action-value function, also known as the Q function, which maps a pair of an action and a state to an expected reward.

The policy is a strategy the agent uses to determine what action to perform given a certain state. An example of a policy would be the epsilon greedy policy. An agent using the epsilon greedy policy will refer to its list of action-state pairs given by the action-value function and with probability ϵ the agent will take a random action and with probability $1 - \epsilon$ will take the action given a state that maximizes its reward. The balance between taking random actions and taking the action with the highest reward is adjusted to find a balance between exploration and exploitation. We want our agent to sometimes explore the environment to find new action-state pairs, but we also want it to learn to exploit the environment to take the best actions to lead to the highest rewards.

Prior to getting into the different reinforcement learning algorithms and their mathematics, I will define an episode and deep reinforcement learning. An episode consists of a sequence of states reaching to the terminal state, the state that ends the game. An episode in a game of chess would be when a check mate occurs. It is important to note that some environments are episodic in nature, like a game of chess, and others are not because they have no clear terminal state, like the stock market. A deep reinforcement learning algorithm is reinforcement learning that uses a neural network as a policy function. The reason neural networks are of high interest as a policy function for reinforcement learning is because a neural network is good at recognizing high level features in data. This makes it useful for when the state space is too large to be completely known.

Reinforcement Learning in Finance

The basic knowledge regarding the general architecture of deep reinforcement learning gained in the last section will allow us to see how it fits in the financial world. Let us associate the financial terms with the deep reinforcement learning terms we learned in the previous section. Our investor is our agent, an investors' goal is to maximize his or her profits, the reward. To do so the investor needs to decide the appropriate time to buy, hold, and sell a certain asset, our action space, based on the current market indicators (i.e., price), the state space. The advantage of a deep reinforcement learning agent is that it can automate the transactions for the investor as well as watch many different data sources at the same time. The agent will be able to know what the state space is at all times, whereas an investor might be busy doing something else, such as eating dinner or fulfilling its other basic human needs. The hope is that the agent will be able to find patterns in the market that will allow it to take advantage of the cryptocurrencies volatility by buying dips and selling when the market is overly bullish. The agent is deemed successful if it can beat the buy-and-hold strategy, which is my current strategy. It is important to take into consideration the transaction fees the agent will incur and remove that from the total reward. In the following section, we will evaluate different algorithms that will be used to teach our agent how to act in the cryptocurrency market.

Section 1.2: Algorithms

Categories of Reinforcement Learning Algorithms

The two categories of reinforcement learning which will be evaluated for this project are value-based algorithms and policy gradient algorithms. A value-based algorithm in deep reinforcement learning is an algorithm that approximates the action-value function using a neural network, the Q-network. This Q-network will output approximations, given state, of which action has the highest expected reward then a policy will be used to choose which action to take (i.e., the epsilon-greedy policy). Value based algorithm are more sample efficient than policy gradient methods, but they require more hyperparameter tunning to work properly [31]. We will be looking at the following value-based algorithms: Deep Q-Learning (DQN), Double DQN (DDQN), Dueling DQN, Gated DQN (GDQN), and Rainbow DQN (RDQN). Policy gradient algorithms in deep reinforcement learning train a neural network to output an action directly. Policy-gradient algorithms will take a state and will return a probability distribution over the possible actions. The primary advantage of policy gradient algorithms is that they can optimize the

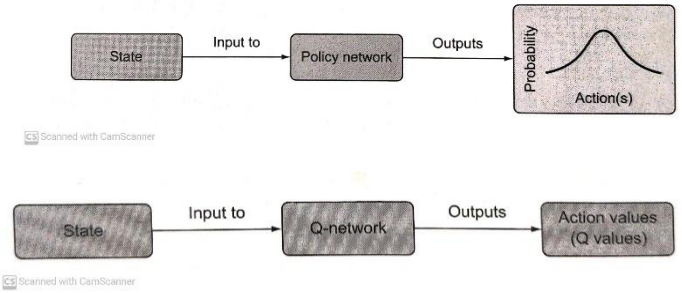

Figure 3.0 Shows the difference between a policy gradient algorithm (top diagram) and a value-based algorithm (bottom diagram). Diagrams are taken from [1] p.90-91.

desired variable while remaining stable [31]. We will be looking at the following policy gradient algorithms: Advantage Actor Critic (A2C), Proximal Policy Optimization (PPO), and Gated Deterministic Policy Gradient (GDPG).

Value-Based Algorithms

Deep Q Learning (DQN)

Deep Q-Learning is the simple case of value-based algorithms when it comes to deep reinforcement learning. It was introduced to the world in a paper published by DeepMind [3]. We will use it as a base model to compare to the other reinforcement learning models and see the performance difference. The general idea of deep Q-learning

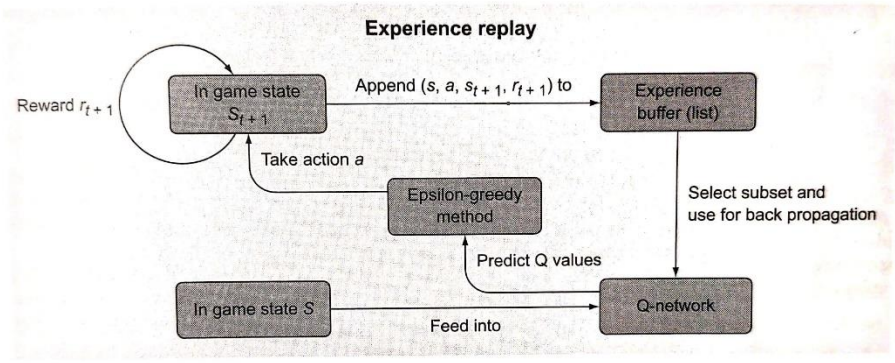

Figure 4.0 A diagram of DQN algorithm using an experience replay component and an epsilon-greedy policy [1] p.82.

is to feed an action and a state to the Q-network which will output a predicted Q-value. The Q-value is the expected reward for the given action-state pair. The action is then taken and the target Q-value as well as the predicted Q-value are used to calculate the loss function (i.e., Mean Squared Error). The Q-network is then updated using a gradient descent method (i.e., Stochastic Gradient Descent). The DQN algorithm is prone to a problem called catastrophic forgetting, which is when we train our Q-network with small batches of data at a time. This causes the new data to be forgotten and thus our agent will not truly be learning to act in the stock market; it will just be basing its decision off the last ones and forgetting everything it previously learned. To solve the issue of catastrophic forgetting, we will add experience replay to our algorithm [1]. Experience replay is a mechanism which will allow our agent to remember what it had previously learned through batch training and thus reducing the noise of our algorithm. To further increase the stability of the DQN algorithm, we will introduce a target Q-network. The target Q-network produces Q-values which are used to train, through backpropagation, the main Q-network. The target Q-network parameters are never trained but they are instead periodically synchronized with the main Q-networks parameters. This lag between the Q-network stops the agent from being overly influenced by one move.

Algorithm [2, p.138]:

1- Initialize parameters for the Q-network, $Q(s,a)$, and target Q-network, $\hat{Q}(s,a)$, with random weights and an empty replay buffer (the list where we are going to store past experiences and use a random subset of these to update the Q-network as opposed to just using the most recent experience to update the network).
2- Use a policy to choose which action to take (i.e., epsilon greedy, with probability ε choose a random action, a, otherwise $a = argmax_a Q(s,a)$ or the softmax policy).
3- Perform action, a, in the environment and observe the next state, s'.
4- Store the experience (s,a,r,s') in the replay buffer.
5- Sample a mini batch of experiences from the replay buffer.
6- For every experience in the buffer, calculate the target $y = r$ if the agent reached the terminal state this step, else $y = r + \gamma max_{a' \in A} Q(s,a)$. The discount factor, γ, determines how much the agent will weigh future rewards as opposed to the more immediate rewards.
7- Calculate loss $L = (Q(s,a) - y)^2$ notice that this loss function uses the computed target network Q-value from step 6.
8- Update Q-network using a gradient descent algorithm (in our case stochastic gradient descent (SGD)) by minimizing the loss in respect to parameter models.
9- Every N-steps copy weights from main Q-network to the target Q-network.
10- Repeat from step 2 until convergence.

Double Deep Q-Learning (DDQN)

DeepMind introduced the DDQN as an improvement to the existing DQN algorithm [4]. A DQN will tend to overestimate Q-values due to the $max_a Q'(s_{t+1}, a)$ part of the target equation, $Q(s,a) = r + \gamma max_a Q'(s_{t+1}, a)$. The paper by DeepMind [4] proposed the solution to change the target Q-values to the following formula $Q(s,a) = r + \gamma max_a Q'(s_{t+1}, argmax_a Q(s_{t+1}, a))$. They prove in their paper that this fixes the issue of overestimation in DQN. Everything in the DDQN algorithm remains the same as DQN aside from the mentioned formula. Recall from the section on DQN, Q' refers to the target Q-network which is a second Q-network which is used to train the primary Q-network Q, as previously mentioned this helps stabilize the algorithm.

Dueling Deep Q-Learning (Dueling DQN)

Another issue with the DQN algorithm which DDQN does not solve is that when an agent experiences a bad state which leads to a low reward instead of lowering the $V(s)$, state-value function which maps state to the expected reward, the agent will store the low reward in its memory due to some action by updating the $Q(s,a)$, the action-value function. The issue with this process is that the agent will keep returning to this unpromising state until all the actions leading to that state are updated in the Q function

[5]. This is computationally inefficient. To solve this issue and increase the algorithms convergence speed, the Dueling DQN was introduced [6].

The DQN will take the features from the Q-network and output a vector of Q-values for each action and then a policy will choose an action to take [2, p.217]. The Dueling DQN will take features from the Q-network and separate them into two different streams. One stream will be predicting the $V(s)$, state-value numbers, and the other will predict the advantage values. The advantage is a computation calculating how much more of a reward an action from a certain state will bring us, $A(s,a) = Q(s,a) - V(s)$. To obtain the Q-value we add the $V(s)$ value to the $A(s,a)$ and subtract the mean value of the advantage given the following formula $Q(s,a) = V(s) + A(s,a) - \frac{1}{N}\sum_k A(s,k)$. The rest of the algorithm is the same as the DQN.

Gated Deep Q-Learning (GDQN)

The Gated DQN algorithm consists of the same algorithm as the DQN and uses Gated Recurrent Unit (GRU) as the policy network. The GRU networks are a type of Recurrent Neural Network (RNN). An RNN is used when context is important. They have sequential memory. The issue with basic RNN's is that they are victims of the vanishing gradient problem, as the network is trained it will be influenced much less by prior events. The GRU network solves this issue by learning to keep relevant information in the model's memory using different gates. We could also have used Long/Short Term Memory networks (LTSM) but these tend to be longer to train than the GRU networks.

Rainbow Deep Q-Learning (RDQN)

The Rainbow Deep Reinforcement Learning (RDQN) is an ensemble algorithm which combines many of the algorithms we have discussed. The RDQN was introduced to try to get the most out of all the improvements to the DQN algorithm [7]. It unites the following algorithms DQN, DDQN, Dueling DQN, Noisy DQN [8], Prioritized Experienced Replay [9], Multi-Step DQN [10], Categorical DQN [11]. This algorithm is computationally costly, but it outperforms the other algorithms when playing the classic Atari video games [7].

Algorithm [5]:
Hyperparameters: $\boldsymbol{B}$ — batch size, $\boldsymbol{V_{max}}$, $\boldsymbol{V_{min}}$, $\boldsymbol{A}$— parameters of support, $\boldsymbol{K}$ — target network update frequency, $\boldsymbol{N}$ — multi-step size, $\boldsymbol{\alpha}$ — degree of prioritized experience replay, $\boldsymbol{B(t)}$ — importance sampling bias correction for prioritized experience replay, $\boldsymbol{\zeta}^*$ — neural network, SGD optimizer.
support grid $z_i = V_{min} + \frac{i}{A-1}(V_{max} - V_{min})$

1- Select input $a = argmax_a \sum_i z_i \, \zeta^*_i(s,a,\theta,\varepsilon), \varepsilon \sim N(0,I)$
2- Observe experience (s,a,r',s')
3- Construct N-step experience $T = (s,a,\sum_{n=0}^{N} \gamma^n \, r^{(n+1)}, s^{(N)})$ and add it to the experience replay with priority $max_T p(T)$.
4- Sample batch from experience replay using probabilities $P(T) \; \alpha \; p(T)^{\alpha}$

5- Compute weights for the batch, where M is the size of the experience replay memory

$$w(T) = (\frac{1}{MP(T)})^{B(t)}$$

6- For each experience from the batch compute the target

$$\varepsilon_1 \varepsilon_2 \sim N(0, I)$$

$$P(y(T) = \bar{r} + \gamma^N z_i) = \zeta^*{}_i(\bar{s}, argmax_{\bar{a}} \sum_i z_i \zeta^*{}_i(\bar{s}, \bar{a}, \theta, \varepsilon_1), \theta^-, \varepsilon_2)$$

7- Project target onto support

8- Update experience priorities $p(T) \leftarrow KL(y(T) \mid\mid Z^*(s, a, \theta, \varepsilon_1), \theta^-, \varepsilon_2$

9- Compute loss: $Loss = \frac{1}{B} \sum_T w(T) p(T)$

10- Make a gradient descent step

11- If $t \; mod \; K = 0$: $\theta^- \leftarrow \theta$

Value Based Algorithms in Automated Trading

The DQN algorithm has been extensively studied in automated trading and it is now often used as a comparison algorithm to test the performance of its improved counterparts [12] [13] [14]. The basic DQN algorithm has been shown to outperform trading strategies such as the buy and hold, decision tree-based method [14] as well as the double cross over strategy [15]. Since these earlier studies, the DQN algorithm has been improved in various ways bringing to life new algorithms such as DDQN and Dueling DQN. Both the DDQN and the Dueling DQN outperform the DQN agent in automated trading [13]. Researchers decided to use Gated Recurrent Unit neural network with the DQN strategy (GDQN), and it outperformed the Turtle trading strategy [16]. Not much has been researched in using the Rainbow DQN (RDQN) as an automated trading agent but one paper published [17] showed that it outperformed the DDQN, DQN as well as the Dueling DQN. Its performance did appear to suffer in highly volatile market conditions which means that it will need quite a bit of adjustments for the cryptocurrency market. It will need to be fed more high-quality data to help it trade in volatile conditions. RDQN greatly outperforms many other value-based algorithms in Atari games [7].

Policy Gradient Algorithms

Advantage Actor Critic (A2C)

The advantage actor critic method is a combination of both policy gradient algorithms and value-based algorithms. The advantage actor critic algorithm has a neural network which returns two vectors: one for the policy and one for the value, $V(s)$. The

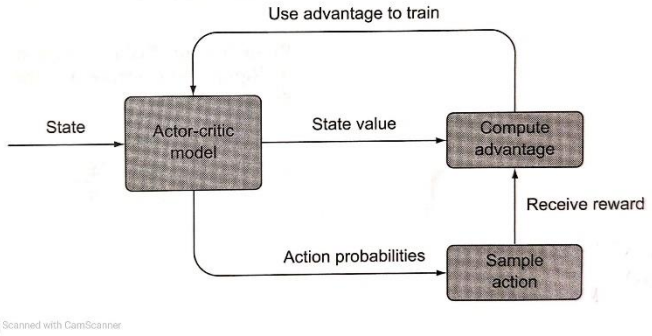

Figure 5.0 A diagram of the Advantage Actor-Critic algorithm [1] p.125.

policy function, the actor, and the value function, the critic, are independent. The policy function is going to return a probability distribution over the possible actions the agent can take in the given state. The value function will output the expected return for the agent given a state and an action. Essentially, the actor will take an action and the critic will inform the actor on the performance of the action. Using the state-value and the action probabilities, the advantage will be calculated, $A(s,a) = Q(s,a) - V(s)$. The advantage value is then used to train the model.

The N-step learning is an addition to the A2C algorithm. It means that the loss is calculated, and the parameters are updated at every N step. With N-step learning instead of updating the neural network at the end of full episodes we update it during the episodes too. This will cause the target value for the critic network to be more accurate and less biased.

Algorithm [18]:

1- Acquire a batch of transitions (s,a,r,s')using the current policy π_θ (either a finite episode or a truncated one).
2- For each state encountered, compute the discounted sum of the next n rewards $\sum_{k=0}^{n} \gamma^k \, r_{t+k+1}$and use the critic to estimate the value of the state encountered n steps later $V(s_{t+n+1})$.

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

3- Update the actor

$$\nabla_\theta J(\theta) = \sum_t \nabla_\theta \, log\pi_\theta(s_t, a_t)(R_t - V(s_t))$$

4- Update the critic to minimize the TD error between the estimated value of a state and its true value.

$$Loss = \sum_t (R_t - V(s_t))^2$$

5- Repeat until convergence

Gated Deterministic Policy Gradient (GDPG)

Deterministic Policy Gradient (DPG) was introduced to tackle control tasks in the continuous environment [19]. It is a type of actor critic method that is deterministic as opposed to stochastic like the A2C. In the A2C algorithm the policy network (actor) outputs the probability of an action given a state $\Pr(action|state)$. The A2C algorithm also computes the actor's expected gradient using this formula $\nabla\theta V(\pi\theta) = Es \sim \rho\pi, a \sim \pi\theta[\nabla\theta log\pi\theta(a|s)Q\pi(s,a)]$.The DGP policy network outputs a single action since it is now deterministic, $\mu\theta(s) = a$. With this action the actor gradient can be calculated using the following formula $\nabla\theta V(\pi\theta) = Es \sim \rho\pi[\nabla\theta\mu\theta(s)\nabla aQ\mu(s,a)|\mu\theta(s) = a]$. As opposed to the A2C algorithm the DPG only integrates relative to the state since the action is deterministic. The Gated Deterministic Policy Gradient (GDPG) is a DPG which uses Gated Recurrent Units for its policy and value neural networks.

Proximal Policy Optimization (PPO)

The PPO alternates between sampling data by acting within the environment and by optimizing the objective function by using Stochastic Gradient Descent (SGD) [20]. The PPO is another type of actor-critic method with a few adjustments. These modifications include a change in the method used to estimate policy gradients and a new way of estimating the advantage. As opposed to taking the log probability of the action as we would in the A2C model, we use the ratio between the new and the old policy scaled by the advantages, $J_\theta = E_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} A_t]$. The advantage function is now updated using the following formula:
$A^{\pi}(T) = r' + \gamma V_\phi^\pi(s') - V_\phi^\pi$.

Algorithm [5]:
Hyperparameters: $\boldsymbol{B}$ — batch size, $\boldsymbol{R}$ — rollout size, **n_epochs** — number of epochs, $\boldsymbol{\varepsilon}$ — clipping parameter, $\boldsymbol{V_\phi^*}$— critic neural network, $\boldsymbol{\pi_\theta}$ — actor neural network, $\boldsymbol{\alpha}$ — critic loss scaling, SGD optimizer.

1- Obtain a roll-out of size R using policy $\pi(\theta)$, storing action probabilities as $\pi^{old}(a|s)$.
2- For each experience T from the roll-out compute advantage estimation:
$$A^{\pi}(T) = r' + \gamma V_\phi^\pi(s') - V_\phi^\pi$$
3- Perform n_epochs passes through roll-out using batches of size B; for each batch:
   a. Compute critic target
   $$y(T) = r' + \gamma V_\phi^\pi(s')$$
   b. Compute critic loss
   $$Loss = \frac{1}{B}\sum_T (y(T) - V_\phi^\pi)^2$$
   c. Compute critic gradients
   $$\nabla critic = \frac{\partial Loss}{\partial \phi}$$
   d. Compute importance sampling weights

$$r_\theta(\mathrm{T}) = \frac{\pi_\theta(\mathrm{a} \mid \mathrm{s})}{\pi^{old}(\mathrm{a} \mid \mathrm{s})}$$

e. Compute clipped importance sampling weights

$$r_\theta^{clip}(\mathrm{T}) = \mathrm{clip}(r_\theta(\mathrm{T}), 1 - \epsilon, 1 + \epsilon)$$

f. Compute actor gradient

$$\nabla^{actor} = \frac{1}{\mathrm{B}} \sum_{T} \nabla_\theta \min(r_\theta(\mathrm{T})\mathrm{A}^\pi(\mathrm{T}), r_\theta^{clip}(\mathrm{T})\mathrm{A}^\pi(\mathrm{T}))$$

g. Make a step of gradient descent using $\nabla^{actor} + \alpha\nabla^{critic}$.

4- Until convergence

Policy Gradient Algorithms in Automated Trading

Policy gradients method are more effective at learning the dynamics of trading than Q-learning, a value-based method [21]. The GDPG algorithm was more stable than the GDQN algorithm for stock trading [17]. Not much research has been done on the PPO algorithm and its application to stock trading. The PPO algorithm has been shown to outperform other online policy gradient methods playing Atari games, therefore it will be worth considering for the trading agent. An ensemble algorithm using the PPO, A2C, DDPG was used to create a trading agent and it outperformed the Dow Jones Industrial Average index (DJIA) and the Min-Variance portfolio allocation strategy. While the ensemble model outperforms the PPO algorithm, the PPO algorithm performs very promisingly [22].

Section 1.3: About The Data

The research published on cryptocurrency trading just use cryptocurrency prices, sentiment analysis and some blockchain information mostly separately. A trading algorithm using all the different data sources has not yet been evaluated. A big advantage with cryptocurrency is the number of Application Programming Interfaces (APIs) which provide a lot of great data sources that can describe the blockchain very well. The following data list was created based on research evaluating the main drivers of cryptocurrency prices as well as what other researchers used for their trading agent. I have also added some data which could be useful to the agent. It is important to note that the data sources may need to be changed because of the evolving landscape of Ethereum. Since Ethereum is moving to a proof of stake concept as its protocol is updated, mining difficulty will become irrelevant since mining will be terminated. A correlation analysis will be done to evaluate which data sources will remain in the model to avoid any collinearity issues which we might run into. The list also contains a link to the location of the data sources.

- **Market Capitalization**: the total dollar value of a company's outstanding shares. (share priced x total outstanding shares) source: [23][24] site: https://etherscan.io/chart/marketcap

- **Daily Transactions**: the number of transactions on the ether network. source: [25] [24] site: https://etherscan.io/chart/tx
- **ERC20 Daily token transfer**: ERC20 is a standard used for creating and issuing smart contracts on the Ethereum blockchain. sub tokens. source: none site: https://etherscan.io/chart/tokenerc-20txns
- **Unique Addresses** : total distinct number of address on the ether network. source: [24] site: https://etherscan.io/chart/address
- **Trading Volume**: number of coins exchanged in a day. Source: [23] [24] site: https://coinmarketcap.com/currencies/ethereum/historical-data/
- **Google Trends:** the amount of searches for the word "Ethereum". source: [25] [26] analysis site: https://trends.google.com/trends/explore?geo=CA&q=ethereum
- **Market Beta:** this will calculate the volatility of Ethereum source: [27] site: this will need to be calculated manually.
- **Average Transaction Fee:** the daily average amount in USD spent per transaction on the Ethereum network. Source: [27] [24] site: https://etherscan.io/chart/avg-txfee-usd
- **Network Difficulty:** the mining difficulty of the Ethereum network. source: [27] [24] site: https://etherscan.io/chart/difficulty
- **Hash Rate:** the measure of the processing power of the Ethereum network. source: [27] [24] site: https://etherscan.io/chart/hashrate
- **Bitcoin Price:** the price of bitcoin source: none site: https://coinmarketcap.com/currencies/bitcoin/
- **Order Book Data:** spread, ask/bid depth, depth difference, ask/bid volume, volume difference, weighted spread, ask/bid slope. source: [28] site: I would need to collect this data over a period of time
- **Tweet Volumes:** the amount of tweets that use the word Ethereum. source: [26] site: https://bitinfocharts.com/comparison/tweets-btc-eth-ltc-doge.html#3y
- **News Sentiment:** the general news sentiment of a crypto news aggregator regarding the crypto market. source: [29] site: https://cryptonews-api.com/
- **Average Block Size:** average block size in bytes of the Ethereum blockchain. source: [24] site: https://etherscan.io/chart/blocksize
- **Network Transaction Fee Chart:** number of ether paid as a fee for the Ethereum network. source: [24] site: https://etherscan.io/chart/transactionfee
- **Bitcoin Hash Rate:** the hash rate of bitcoin. source: none site: https://www.blockchain.com/charts/hash-rate
- **Macro Economics:** S&P500, Eurostoxx, DOW30, NASDAQ, Crude oil, SSE, Gold, VIX, Nikkei225, FTSE100. Source: [24] site: unknown
- **Daily Active Ethereum Address:** unique addresses that were on the Ethereum network as senders or receivers source: none site: https://etherscan.io/chart/active-address

Chapter 2: Project Results

Section 2.1: Final Algorithm, Trading Results, and Graphs

CNN with Shared Model Proximal Policy Optimization (PPO)

The final model chosen for the trading agent was a proximal policy optimization algorithm with a shared neural network model. The inner workings of the algorithm have been looked at in chapter 1. The PPO algorithm was chosen due to the problems it solved against some of the other algorithms mentioned in chapter 1. Other policy gradient methods tend to be sensitive to the choice of step size, they have a lot of noise, and they tend to have catastrophic drop in performance [30]. The biggest reason for this being the algorithm of choice for this project is because it has better sample efficiency and does not require millions or even billions of timesteps to learn a task [30].

The shared part of the model implies the data passes through a shared neural network prior to going through the actor and the critic branches of the algorithm. The architecture for the model can be seen in figure 2.1. The purpose of the neural network linking both

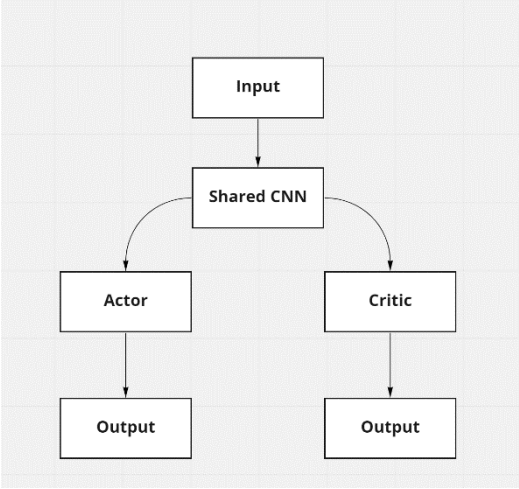

Figure 2.1 Algorithm architecture for trading agent.

the actor and the critic models is for feature selection. Due to the amount of features we are feeding the algorithm, the shared neural network is used to select the most valuable features for the model.

The use of convolutional neural networks (CNN) as the policy networks was initially meant to be a basis to test the performance of the algorithm using recurrent neural networks (RNN). Due to the lack of computing power in my possession, it took too long to attempt to train an algorithm with LTSM neural networks. Therefore, it is due to the convergence time of the CNN which they have been use in the model. Future work might consider using the gated recurrent neural network (GRU) which generally converges faster than the LTSM networks.

The reward function calculates the profit or loss when the agent sells i.e., reward = buy price – sell price.

Results

The model was trained for 20,000 episodes and showed good results of having learned to train Ethereum. Episode one results can be seen in figure 2.2. We can see that the agent is quickly losing all its initial balance and making poor trading decisions. As it learns, it slowly improves its trade timing and has no losing trades. Recall that the agent gets a greater reward by having a greater profit after a trade. As seen in figure 2.3 the agent starts making less trades and at more sparse intervals. In Figure 2.4 we see the agent making very few trades but now being able to identify the market peak and estimate the crashes relatively well. Once the agent was trained, we tested it on a 500-day period of data an unseen data set. The results of the test can be seen in figure 2.5. During

Figure 2.2 episode 1

Figure 2.3 episode 692

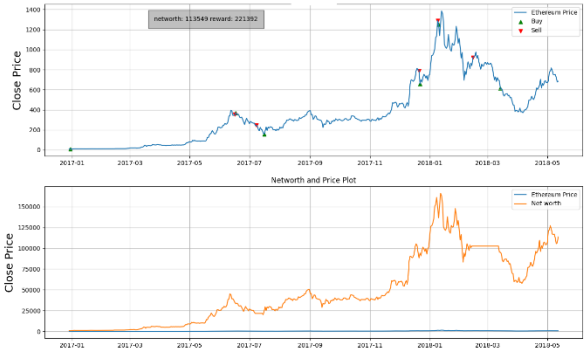

Figure 2.4 episode 19 436

Figure 2.2, 2.3, 2.4 and 2.5 show the price of Ethereum in the top plot with red marker representing sell orders and green markers representing buy orders. The bottom plot on the figures shows the agent's net worth in orange and the Ethereum price in blue.

testing the agent accumulated a total net worth of $6,457.00 and a reward of 5633. In the final best training episode, the agent is able to find the market highs and low relatively well. The agent makes no losing trades and trades much less than in the early training episodes. Looking at the episode orders for the first 10k episodes in figure 2.7 we see that the number of trades performed by the agent drops very quickly and converges to 20. This makes it quite surprising that it only performs 3 trades on the testing data when it had seemed to find an optimal trade point of 20 during training. The agent's training is

shown to have worked looking at figure 2.8. We can observe the rise of the net worth as more training episodes go by.

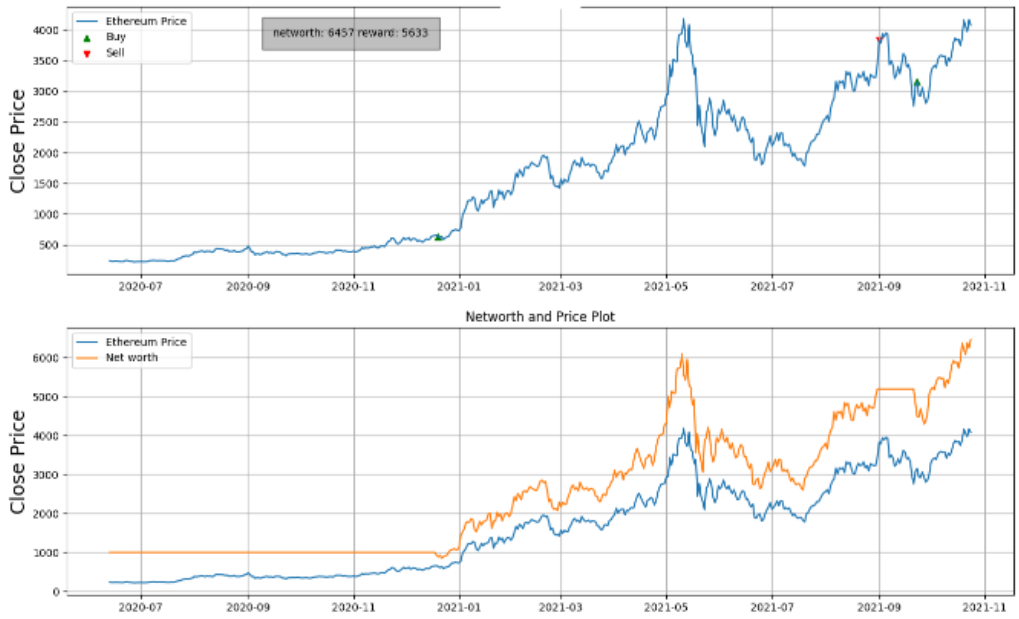

Figure 2.5 agent performance on test data

Figure 2.6 Plot of amount of orders per episode

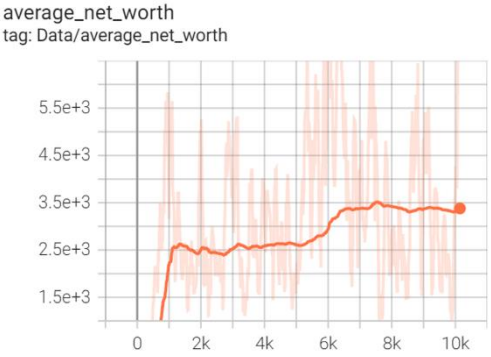

Figure 2.7 Plot of agent net worth per episode

Section 2.2: Challenges and Future Steps

Despite the agent not making any losing trades it does not beat the market, which was the goal of the project. If the agent would have bought and held the currency it would have had a greater profit, because of this it did not perform as well as expected. There are some adjustments that could be made to attempt to increase the agent's performance. The

first would be to trade hourly as opposed to daily. Allowing the agent to make more trades on an hourly basis would make many smaller positive profits which could help it increase its profit. We did not dot this because hourly data was not available for most of the data gathered for the analysis .Furthermore 20,000 episodes is not a long time to train an agent. Unfortunately, I do not have the computing power to train on more episodes. In an attempt to train on 100,000 episodes, the trading agent hit a local optimum causing it to only buy once. This could be because the agent observed that buying and holding was an optimal strategy. This local optimum could be overcome by training the agent for a much longer amount of time but due to the lack of computing power, I could not test this hypothesis. A lot of time was spent creating the necessary coding package to test the agent. In an ideal scenario, I would have been able to test multiple different algorithms as well as an ensemble of algorithms to create the best trading agent. The project overall did not succeed in beating the buy-and-hold strategy, but it was able to trade positively and to improve its performance, which is good.

Chapter 3: Python Code Documentation

A great deal of the structure of the code was built from the foundations of a YouTube tutorial series created by Pylessons which can be found on his YouTube channel https://www.youtube.com/channel/UCZ8qczwOOrL8ywaCRz5WUjw . He also goes into multiprocessing which is very useful for accelerating the agents training process. The goal of this chapter is to properly document and explain the code found in the GitHub project with the intention of allowing whoever chooses to modify and adjust the project as they see fit.

Section 3.1: Project Files and Folders

The code triggers a lot of automated processes which will save training results, testing results and create visualizations both using pyplots as well as through tensor board. These automations might seem to clutter the project with a lot of information but is useful for understanding the agent's performance to identify issues and improvement points. Some of these folders and files will need to be cleaned and organized manually before each run. The folders and files which are automatically created are the following:

1. "2022_01_14_11_01_Crypto_trader": The name of the folder will change depending on the date it is created but it will always end with "_Crypto_Trader". This folder contains tensorboard information of the best models in the form of "198214.92_Crypto_trader_Actor.h5" and "198214.92_Crypto_trader_Critic.h5". Each time a new and better training episode is encountered the model parameters are saved. When going back to test the agent you can choose which strategy you would like the agent to use from its previous training. You will usually choose the model with the highest reward number. This folder also contains a log.txt file as well as a Parameters.txt file. As you train different strategies you will have a greater number of folders and these last two files help you remember what the parameters for the model were. The Parameters.txt will contain the following information:

2022_01_14_11_01_Crypto_trader > Parameters.txt

```
1   training start: 2022-01-14 11:01
2   initial_balance: 1000
3   training episodes: 400000
4   lookback_window_size: 50
5   lr: 1e-05
6   epochs: 10
7   batch size: 64
8   normalize_value: 40000
9   model: CNN
10
```

Figure 3.1 parameters.txt file content

These are the parameters chosen for the training session. The log.txt file contains the following information about the model being saved (the model gets saved when it has a new max reward): current time, average net worth, number of trades performed in the episode, actor loss, critic loss. This is further information to use to be able to track the agent's performance.

2. "runs": This folder contains information kept from each training sessions for the purpose of visualizing information in tensorboard. Once the code has finished running you will run the following line in the terminal: "tensorboard --logdir runs" you will then be able to open the tensorboard visualizer containing all of the sessions that were saved in the run folder. The tensorboard visualizer will open in a local host and will look like in figure 3.2.

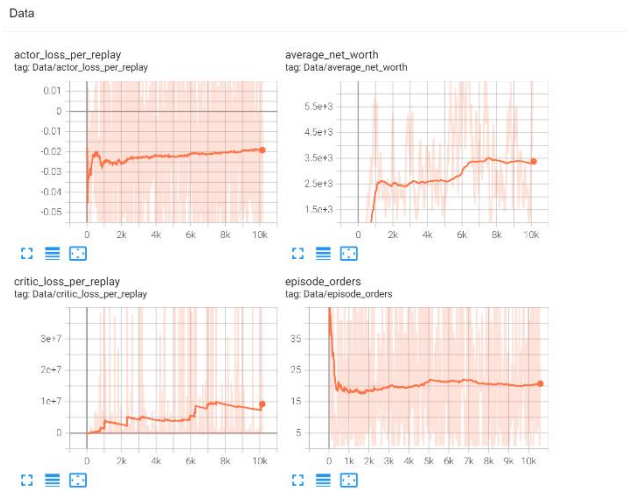

Figure 3.2 view of the tensoboard visualizer graphs.

With the default version of the visualization function you will be able to see the actor loss per replay, the agents average net worth, the critic loss per replay and the number of orders performed by the agent per episode (the number of trades the agent made). Tensorboard has multiple great features including the ability to remove and add the models you would like plotted in the graphs as well as a slider to smooth the curves for better trend visualization.

3. “test_results.txt”: This folder will store all the runs performed on the test data. It will include the current date, the model being used for testing i.e. “198214.92_Crypto_trader” if we are using the above trained model, the number of test episodes, the final net worth after the test episode, the number of orders per episode, the number of no profit episodes as well as the neural network used for the model, “CNN” for convolutional neural network.
4. “train_trades_plot_episode_0.png”: As the agent is training it will save images every time a best new model is found, a model which has accumulated the highest reward out of the current training session. This allows the developer to see the agents progress as it is running to get an idea of how the training is going. The image will look like the following:

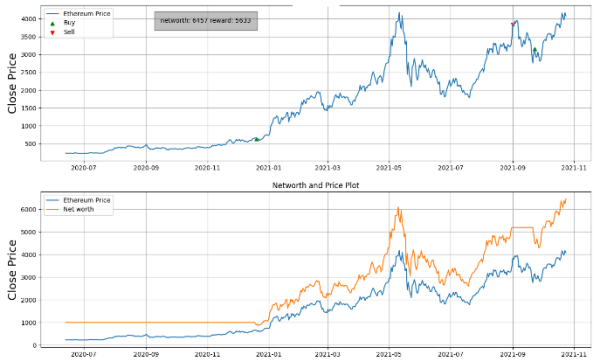

Figure 3.3 figure of training performance. It shows the price of Ethereum in the top plot with red marker representing sell orders and green markers representing buy orders. The bottom plot on the figures shows the agent’s net worth in orange and the Ethereum price in blue.

It is important to save these images in a separate folder and then delete them before every training session, this will avoid confusions when trying to compare the progress image of different training sessions. If this is not done you will not know which image refers to which model and furthermore the images have a chance of being written over with a new output causing further confusion. “trades_plot_episode_0.png” this is the same image outputted as “train_trades_plot_episode_0.png” but for testing runs.

Aside from the standard .gitignore and README.md files the project contains a “cryptoanalysis_data.csv” file. This file contains cleaned data which I put together and used for my trading agent. The default version of the code will use this data to train the agent. Information about the data can be found in the “About The Data” section of the documentation. Modifications to multiple functions need to be made for the agent to train on other data. To learn how to modify the code to change the data used in the environment you can look under the “Customizing Environment Data” section of this chapter.

Section 3.2: Env.py

This file contains two different classes. The first being the CustomAgent() class which contains most of the functions allowing the agent to save models and log important information for evaluation. It also dictates the agent's inner workings, such as predicting an action to take. The functions within it will not need to be adjusted because all the hyperparameters can be chosen as arguments while calling onto the class. If you wanted to use a non-actor critic algorithm, you would need to change a couple of functions within the CustomAgent() class. These include: the replay() function which will calculate actor and critic loss, the save() function and the load() function. In changing the model to anything other than an actor critic you would need only save and load the weights for a single model. The second class found within the env.py file is the EthereumEnv() class. This is where the inner working of the environment is situated and it will need to be modified to use a different dataset then the one provided in "crypto_analysis.csv". This is also the class where you can modify the reward function. If you were to modify the data being inputted to the environment would require to modify the functions within the class EthereumEnv(): reset(), _next_observation() as well as within CustomAgent() class in the __init__() portion the line self.state_size = (look_back_window_size, 20) will need to be modified. The 20 refers to the number of variables within the dataset being fed into the environment and so if you were to add or remove variables (columns) from the "crypto_analysis.csv" file this number would need to be modified accordingly. Customizing environment data is covered more thoroughly in the Customizing Environment Data portion of the documentation.

Class CustomAgent()

Description: Use this function to initialize an agent to train and test. This function will output and env.CustomAgent type. The output of this function is used in the train_agent() and test_agent() functions which run the reinforcement learning algorithm.

Usage:

```
CustomAgent(lookback_window_size=lookback_window_size, lr, epochs, optimizer, batch_size, model)
```

Arguments:

lookback_window_size: integer of the number of candles we want the agent to have access to.

batch_size: int of the size of the buffer replay list.

lr: int of the learning rate for the model.

epochs: int of the number of passes to complete to train the data the models

optimizer: this optimizer is imported from tensorflow.keras.optimizers. The available optimizers are therefore whatever is imported from that module. An optimizer is used to find the value of weight which has the lowest. Another example of an optimizer would be a gradient descent algorithm.

model: a string of the model type you would like for the neural net. The are two options: “CNN” for convolutional neural networks and “LTSM” for long short term memory. “***”Dense”*** is also available.

Example:

```
agent = CustomAgent(lookback_window_size=lookback_window_size,
                    lr=0.00001, epochs=10, optimizer=Adam, batch_size=64,
model="CNN")
```

Class EthereumEnv()

Description: Use this function to initialize a trading environment to train and test an agent. This function will output and env.EthereumEnv type. The output of this function is used in the train_agent() and test_agent() functions which run the reinforcement learning algorithm.

Usage:

```
    train_env = EthereumEnv((self, df, initial_balance=1000,
lookback_window_size=50, trading_fee=0.07, debug_mode=False)
```

Arguments:

df: cleaned pandas data frame with historical Ethereum data.

initial_balance: integer of the starting balance for the agent to trade with. The default is 1000.

lookback_window_size: integer of the number of candles we want the agent to have access to

trading_fee: the fee applied to each transaction. The default is based on the binance fee which is 0.075.

debug_mode: a Boolean value. If debug mode is set to True a text file will be created containing transaction date and the amount of crypto held by the agent after the transaction. This can be used to make sure the agent is trading the proper amount of the currency. In the default case we want the agent to buy or sell of its crypto held during each transaction.

Example:

```
lookback_window_size = 50
train_env = EthereumEnv(
    train_df, lookback_window_size=lookback_window_size)
```

Section 3.3: Models.py

This file contains four main functions with a handful of helper functions. The first important function which is Random_games(). The Random_games() function is useful for seeing how an agent would trade randomly. The agent's random performance can be compared to the chosen trading strategy to see how much better a strategy is against random selection. The file contains an Actor and Critic model which have both been archived in exchange for the Shared_Model() which uses both the actor critic and combines them with a shared neural network. For more information on the shared model refer to section 2.1. If the user chooses not to use the shared model for their analysis they will need to add the new algorithm to this file and call it in the CustomAgent() class. The user will need to change the self.Actor = Shared_Model(…) to their desired model. The two other important functions in this python file are train_agent() and test_agent(). Train_agent() is the function used to train the agent. It will save the best episodes, the episodes which have the highest reward, which will modify the way the agent trades in future episodes. The test_agent() function is used to test the agents performance on a test dataset and is written in a similar fashion to the train_agent() function.

Class Random_games()

Description: This function executes a trading agent which will buy and sell currency at random. It is useful for demonstrating the utility of an "intelligent" algorithm.

Usage:

```
Random_games(env, visualize, train_episodes=50, training_batch_size=1000)
```

Arguments:

env: the environment output given by EthereumEnv(…) function.

visualize: a Boolean value where if True outputs the results of each episode as agent is running. The default value is False.

train_episode: integer of the number of episodes the agent will use to train. The default is 50.

training_batch_size: integer of the maximum amount of steps per episode. The default is 1000.

Example:

```
Random_games(train_env, visualize=False,
train_episodes=10)
```

Class Shared_Model()

Description: This function runs the algorithm for the shared model. Refer to section 2.1 for more information on the shared model used for the project.

About the model: PPO is a actor critc model (A2C) with a handful of changes:

1/ Trains by using a small batch of expiriences. The batch is used to update the policy. A new batch is then sampled and the process continues. This slowly changes the policy to a better version.

2/ New formula to estimate policy gradient. Now uses the ratio between between the new and the old policy scaled by the advantage.

3/ New formula for estimating advantage.

This function is called within the CustomAgent() class.

Usage:

```
Shared_Model(input_shape, action_space, lr, optimizer, model="Dense")
```

Arguments:

input_shape: a tuple with two digits contain the length of the data i.e. the lookback window size and the number of variables in the data. This is declared in CustomAgent(…) as self.state_size in the __init__(…) funcition and would need to be modified if the number of variables given to the environment changes.

action_space: integer of the number of actions possible that the agent can take. This number will always be three for buy, sell and hold.

lr: int of the learning rate for the model.

optimizer: this optimizer is imported from tensorflow.keras.optimizers. The available optimizers are therefore whatever is imported from that module. An optimizer is used to find the value of weight which has the lowest. Another example of an optimizer would be a gradient descent algorithm.

model: a string of the model type you would like for the neural net. The are two options: “CNN” for convolutional neural networks and “LTSM” for long short term memory.

Example:

```
self.Actor = self.Critic = Shared_Model(
        input_shape=self.state_size, action_space=self.action_space.shape[0], lr=self.lr,
optimizer=self.optimizer, model=self.model)
```

Function train_agent()

Description: This function is used to run the reinforcement learning algorithm and save the parameters that leads to the greatest reward each time a greater reward is found.

Usage:

```
train_agent(env, agent, visualize=False, train_episodes=50, training_batch_size=500)
```

Arguments:

env: the environment output given by EthereumEnv(…) function.

agent: the output given by CustomAgent(…) function.

visualize: a Boolean value where if True outputs the results of each episode as agent is running. The default value is False.

train_episode: integer of the number of episodes the agent will use to train. The default is 50.

training_batch_size: integer of the maximum amount of steps per episode. The default is 500.

Example:

```
agent = CustomAgent(lookback_window_size=lookback_window_size,
                lr=0.00001, epochs=10, optimizer=Adam, batch_size=64, model="CNN")
train_env = EthereumEnv(
    train_df, lookback_window_size=lookback_window_size)
train_agent(train_env, agent, visualize=False,
          train_episodes=400000, training_batch_size=500)
```

Function test_agent()

Description: This function is used to run the reinforcement learning algorithm on test data. It uses a given saved model from resulting from train_data() and uses the model on the given test data. For more information on the parameter saving process and automated folder creation review Section 3.1.

Usage:

```
test_agent(env, agent, visualize=True, test_episodes=10, folder="",
name="Crypto_trader")
```

Arguments:

env: the environment output given by EthereumEnv(…) function.

agent: the output given by CustomAgent(…) function.

visualize: a Boolean value where if True outputs the results of each episode as agent is running. The default value is False.

test_episode: integer of the number of episodes the agent will use to train. The default is 10.

folder: folder created by the train_agent(…) where the model is saved which the user would like to use for the test data.

name: file created by the train_agent(…) found in the folder stated in the folder argument of the function where the model is saved which the user would like to use for the test data.

Example:

```
  agent = CustomAgent(lookback_window_size=lookback_window_size,
                      lr=0.00001, epochs=1, optimizer=Adam, batch_size=128,
model="LSTM")
    test_env = EthereumEnv(
        test_df, lookback_window_size=lookback_window_size)
    test_agent(test_env, agent, visualize=False, test_episodes=1,
               folder="2022_01_18_10_40_Crypto_trader",
name="122580.55_Crypto_trader", comment="")
```

Section 3.4:Utils.py

This file contains two functions one which saves order history to a text file and another which plots the agent's performance when the best episodes are saved. The performance plots can be seen in figure 3.3. The first plot in the figure is the Ethereum closing price during the saved episode with the buy orders as green triangles and the sell orders as red triangles. The second plot in the figure is the agent's net worth as an orange line and the Ethereum for the episode as a blue line.

Function Write_to_file()

Description: This function is used to log the order history each time a buy or sell order is completed during the episode of training session. This is useful for seeing the metrics of the bot performance. These metrics include agent balance, net worth, crypto bought, and crypto sold. This function is currently only ran if debug_mode is set to True for CustomAgent(…).

Usage:

```
Write_to_file(Date, net_worth, filename='{}.txt'.format(datetime.now().strftime("%Y-%m-%d %H-%M-%S")))
```

Arguments:

Date: The date of the transaction completed in the episode.

net_worth: integer of the agent net worth at the time the transaction was completed.

filename: a string of the name for the txt file to be saved. The filename will always contain the current date and time too.

Example:

```
Write_to_file(Date, self.net_worth)
```

Function performance_plots()

Description: This function is used to create figures containing two plots of the bots performance.

Usage:

```
performance_plots(avg_reward, net_worth, n_episodes)
```

Arguments:

avg_reward: list of average reward per episode.

net_worth: list of final net worth per episode.

n_episode: integer of number of episodes executed.

Example:

```
performance_plots(avg_episode_reward_list,
                  net_worth_list, train_episodes)
```

Section 3.5: Main.py

This file is the main file which gets ran to put the agent into action. This file cleans the data that is being used for the agent, creates the agent as well as the environment, then trains the agent and tests the agent. This file does not contain any function but uses the functions from the other files.

Section 3.6: Customizing Environment Data

This section will explain step by step how to change the functions so that they can use different environment data.

1. In the main.py file I am currently importing cryptoanalysis_data.csv, this will need to be changed to the path of your data set. You will then need to clean and standardize the dataset as you see fit.
2. Once this is completed you will need to go into the env.py file and under the class CustomAgent(…) in the __init__(…) function you will need to change the self.state_size = (lookback_window_size, 20). The 20 represents the number of columns in the given dataset. For the default data, cryptoanalysis_data.csv, the are 20 columns.
3. Again in the env.py file, you will now go under EthereumEnv(…) class in the reset(…) function and change the self.blockchain_data.append([…]) line with the column names in your chosen dataset. The current names such as “receive_count” refer to column names in the cryptoanalysis_data.csv file. You will also need to modify the _next_observation(…) function in the same manner. This function also has a self.blockchain_data.append([…]) line which is currently referencing the column names from the cryptoanalysis_data.csv dataset.

Chapter 4: Trading Dashboard and User interface

Section 4.1: Customizing Environment Data

The trading dashboard software allows you to host a trading agent on a server and manage the agent using the dashboard from another server. I personally use a raspberry pi to run my agent and I manage the agent through the dashboard which I open using local host which calls, using an API, to the raspberry pi. This allows me to keep my bot running on my raspberry pi 24/7 and still be able to run other software as well as turn off my main computer. This software is designed to work with the Binance trading platform. The steps to install and running the software are as follows:

1. Clone the GitHub repository of the software: https://github.com/roblen001/Crypto-Bot
2. The dashboard will not work in its default state. You must edit some of the code prior to having it in a working format.

a. In the newscraper.py file you must change all the occurrences of “headers =” to your personal computer user agent. This information can be found by searching “my user agent in chrome”.
b. Now you will need to create a .env file in the flask-api folder with the following information:

   API_KEY= FILL

   API_SECRET= FILL

   DEV_EMAIL = FILL
   EMAIL_PASS = FILL

   PERSONAL_EMAIL = FILL

   TAAPI = FILL

   API_KEY and API_SECRET need to be filled with you binance account apis. The DEV_EMAIL is the email address I use to automatically send an email to my PERSONAL_EMAIL account. The EMAIL_PASSWORD is for the DEV_EMAIL. TAAPI is a token created from https://taapi.io/ it is used for the default RSI trading strategy used by the software.

3. Once the above has been done the software can now be started. In a terminal redirect your working directory to the flask-api folder and run the line “flask start”. Then in a new terminal redirect the working directory to the front-end folder and run the line “gatsby develop”.

4. You should now be able to access the dashboard on your local host.

Section 4.2: User Interface Usage Guide

Figure 4.1 trading dashboard interface.

The dashboard has been organized in a grid lay out with three columns and two rows. The top left section tracks the agent transaction history. It will track the agent's buy and sell orders the moment it performs an order. You can refer to this section to keep track of the agent's trading point. Each time the agent buys or sell a currency, you will also be able to receive an email that a transaction has occurred. The server running using flask also scrapes a crypto news aggregator 24/7. In the bottom left section, you can see news article titles. The news can be filtered by latest or top. By clicking on an article component, you will be redirected to the full news article. The top middle portion contains a candle chart as well as different price change statistics regarding a currency. The information can be changed to different currencies by clicking on different options in the drop-down menu. The bottom middle portion contains a chart with a different information regarding the top currencies. The top right portion of the dashboard is a current work around to being able to track the amount of money given to the agent. To be able to calculate total profits or loss I needed to keep track of the amount of money that has been given to the agent. There is no API call that does this in the Binance API at the time of writing. Therefore, each time you send USDT, a stable coin currency, to the Binance account you must input the amount that was giving. The user will be able to see what dates the bot was giving money and how much was given in USDT. The bottom right side contain user balance information, profit, agent results and button to start and stop the agent from trading. The drop-down menu lets the user choose which currency it wants the agent to be trading. Once the desired currency is selected the start button can be pressed for the agent to start trading. The agent will then buy the selected currency at the opportune moment according to the strategy being used. The stop button can be used to stop the agent from trading the currency. Note that the stop button will not force the agent to sell, the currency will remain in the state of the last action that occurred. If the agent sold last, then the stop will cause the agent not to buy again and if the agent bought last then the stop button will prevent the agent from selling. Net profits refer to the profits or

losses the agent took from the given money. If the agent was given 1000 USDT and currently has a valuation of 1200 USDT then the net profit statistic will display 200$. Percent positive trades refer to the percent of positive trades the agent has made. A positive trade is a trade where the agent sells the currency for more than it bought the currency. Bot vs buy and hold refers to the agent's performance against the buy and hold trading strategy. It will subtract the users total account net worth by the users valuation if they had just bought and held the chosen currency. The account balance shows all the currencies being held in the users account and their price change over the last 24 hours.

Section 4.3: Modifying The Trading Strategy

The trading agent runs in the bot.py which can be found in the flask-api folder. To change the default RSI trading strategy to the desired strategy the user will need to modify the on_message(…) function.

References

[1] Brown, Brandon, and Alexander Zai. *Deep Reinforcement Learning in Action*. Shelter Island, NY: Manning Publications Company, 2020.

[2] Lapan, Maxim. *Deep Reinforcement Learning Hands-on: Apply Modern RL Methods to Practical Problems of Chatbots, Robotics, Discrete Optimization, Web Automation, and More*. Birmingham: Packt, 2020.

[3] Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wiestra, and Martin Riedmiller. "Playing Atari with Deep Reinforcement Learning." *DeepMind Technologies*, December 19, 2013. https://arxiv.org/abs/1312.5602v1.

[4] van Hasselt, Hado, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning." *Thirtieth AAAI Conference on Artificial Intelligence* 30, no. 1 (March 2, 2016). https://doi.org/https://ojs.aaai.org/index.php/AAAI/article/view/10295.

[5] Ivanov, Sergey, and Alexander D'yakonov. "Modern Deep Reinforcement Learning Algorithms," July 6, 2019. https://doi.org/https://arxiv.org/abs/1906.10025.

[6] Wang, Ziyu, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. "Dueling Network Architectures for Deep Reinforcement Learning." *Proceedings of The 33rd International Conference on Machine Learning* 48 (June 20, 2016): 1995–2003. https://doi.org/https://proceedings.mlr.press/v48/wangf16.html.

[7] Hessel, Matteo, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. "Rainbow: Combining Improvements in Deep Reinforcement Learning." *Thirty-Second AAAI Conference on Artificial Intelligence*, April 2018. https://doi.org/https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/viewPaper/17204.

[8] Fortunato, Meire, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Matteo Hessel, Ian Osband, Alex Graves, et al. "Noisy Networks for Exploration," July 9, 2019. https://doi.org/https://arxiv.org/abs/1706.10295.

[9] Schaul, Tom, John Quan, Ioannis Antonoglou, and David Silver. "Prioritized Experience Replay," February 25, 2016. https://doi.org/https://arxiv.org/abs/1511.05952.

[10] Sutton, Richard S. "Learning to Predict by the Methods of Temporal Differences." *Machine Learning* 3, no. 1 (1988): 9–44. https://doi.org/10.1007/bf00115009.

[11] Bellemare, Marc G, Will Dabney, and Remi Munos. “A Distributional Perspective on Reinforcement Learning.” *Proceedings of the 34th International Conference on Machine Learning* 70 (August 6, 2017): 449–58. https://doi.org/https://proceedings.mlr.press/v70/bellemare17a.html.

[12] Lucarelli, Giorgio, and Matteo Borrotti. “A Deep Reinforcement Learning Approach for Automated Cryptocurrency Trading.” *IFIP Advances in Information and Communication Technology*, 2019, 247–58. https://doi.org/10.1007/978-3-030-19823-7_20.

[13] Li, Yuming, Pin Ni, and Victor Chang. “Application of Deep Reinforcement Learning in Stock Trading Strategies and Stock Forecasting.” *Computing* 102, no. 6 (2019): 1305–22. https://doi.org/10.1007/s00607-019-00773-w.

[14] Chakole, Jagdish Bhagwan, Mugdha S. Kolhe, Grishma D. Mahapurush, Anushka Yadav, and Manish P. Kurhekar. “A Q-Learning Agent for Automated Trading in Equity Stock Markets.” *Expert Systems with Applications* 163 (2021): 113761. https://doi.org/10.1016/j.eswa.2020.113761.

[15] Sattarov, Otabek, Azamjon Muminov, Cheol Won Lee, Hyun Kyu Kang, Ryumduck Oh, Junho Ahn, Hyung Jun Oh, and Heung Seok Jeon. “Recommending Cryptocurrency Trading Points with Deep Reinforcement Learning Approach.” *Applied Sciences* 10, no. 4 (2020): 1506. https://doi.org/10.3390/app10041506.

[16] Wu, Xing, Haolei Chen, Jianjia Wang, Luigi Troiano, Vincenzo Loia, and Hamido Fujita. “Adaptive Stock Trading Strategies with Deep Reinforcement Learning Methods.” *Information Sciences* 538 (2020): 142–58. https://doi.org/10.1016/j.ins.2020.05.066.

[17] Wu, Xing, Haolei Chen, Jianjia Wang, Luigi Troiano, Vincenzo Loia, and Hamido Fujita. “Adaptive Stock Trading Strategies with Deep Reinforcement Learning Methods.” *Information Sciences* 538 (2020): 142–58. https://doi.org/10.1016/j.ins.2020.05.066.

[18] Vitay, Julien. Deep reinforcement learning. Accessed October 5, 2021. https://julien-vitay.net/deeprl/ActorCritic.html.

[19] Lillicrap, Timothy P, Jonathan J Hunt∗, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. “Continuous Control with Deep Reinforcement Learning.” *conference paper at ICLR 2016*, July 5, 2016. https://doi.org/https://arxiv.org/abs/1509.02971.

[20] Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. “Proximal Policy Optimization Algorithms,” August 28, 2017. https://doi.org/https://arxiv.org/abs/1707.06347.

[21] Sadighian, Jonathan. “Deep Reinforcement Learning in Cryptocurrency Market Making,” November 21, 2019. https://doi.org/https://arxiv.org/abs/1911.08647.

[22] Yang, Hongyang, Xiao-Yang Liu, Shan Zhong, and Anwar Walid. “Deep Reinforcement Learning for Automated Stock Trading: An Ensemble Strategy.” *SSRN Electronic Journal*, 2020. https://doi.org/10.2139/ssrn.3690996.

[23] Alessandretti, Laura, Abeer ElBahrawy, Luca Maria Aiello, and Andrea Baronchelli. “Anticipating Cryptocurrency Prices Using Machine Learning.” *Complexity* 2018 (2018): 1–16. https://doi.org/10.1155/2018/8983590.

[24] Jang, Huisu, and Jaewook Lee. “An Empirical Study on Modeling and Prediction of Bitcoin Prices with Bayesian Neural Networks Based on Blockchain Information.” *IEEE Access* 6 (2018): 5427–37. https://doi.org/10.1109/access.2017.2779181.

[25] Kristoufek, Ladislav. “What Are the Main Drivers of the Bitcoin Price? Evidence from Wavelet Coherence Analysis.” *PLOS ONE* 10, no. 4 (2015). https://doi.org/10.1371/journal.pone.0123923.

[26] Abraham, Jethin, Daniel Higdon, John Nelson, and Juan Ibarra. “Cryptocurrency Price Prediction Using Tweet Volumes and Sentiment Analysis.” *SMU Data Science Review* 1, no. 3 (2018). https://doi.org/https://scholar.smu.edu/datasciencereview/vol1/iss3/1/.

[27] Sovbetov, Yhlas. “Factors Influencing Cryptocurrency Prices: Evidence from Bitcoin, Ethereum, Dash, Litcoin, and Monero.” *Journal of Economics and Financial Analysis* 2, no. 2 (2018): 1–27. https://doi.org/https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3125347.

[28] Guo, Tian, Albert Bifet, and Nino Antulov-Fantulin. “Bitcoin Volatility Forecasting with a Glimpse into Buy and Sell Orders.” *2018 IEEE International Conference on Data Mining (ICDM)*, 2018. https://doi.org/10.1109/icdm.2018.00123.

[29] Vo, Anh-Dung. “Sentiment Analysis of News for Effective Cryptocurrency Price Prediction.” *International Journal of Knowledge Engineering* 5, no. 2 (2019): 47–52. https://doi.org/10.18178/ijke.2019.5.2.116.

[30] Schulman, John. “Proximal Policy Optimization.” OpenAI. OpenAI, September 2, 2020. https://openai.com/blog/openai-baselines-ppo/.

[31] https://proceedings.neurips.cc/paper/2017/file/facf9f743b083008a894eee7baa16469-Paper.pdf