Portfolio Allocation and Reinforcement Learning

René Garcia and Alissa Marinenko

December 3, 2022

Abstract

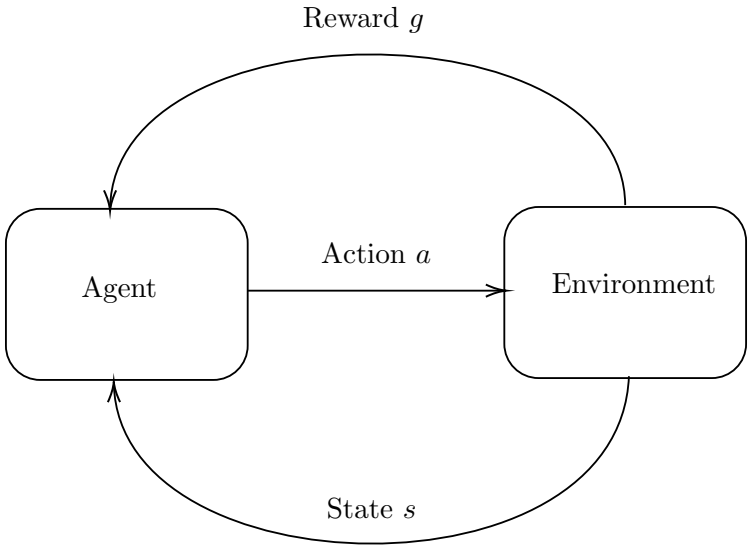
In this chapter, we review briefly the methodology of reinforcement learning and describe its application to the financial problem of portfolio allocation. In this context, we define the environment as a set of states, captured by such financial variables as stock returns or technical indicators, and of actions, mainly the determination of wealth shares to invest in each asset. Optimal value functions are obtained through the Bellman optimality equation, a well-established principle in both reinforcement learning and portfolio optimization. Deep reinforcement learning algorithms have the advantage to provide approximate solutions since most portfolio problems lack analytical solutions. We describe several algorithms and apply them to classical portfolio allocation problems, where risk minimization and return maximization are combined, with or without accounting for trading costs.

Contents

1 Reinforcement Learning

1.1 Introduction

Reinforcement learning is a branch of machine learning that deals with learning control strategies by interacting with a complex environment. At the heart of this framework, there is an agent interacting with an environment at each step/instant in time.



In the above diagram, at every time-step, the agent observes an environment described by a state s (e.g. current stock prices, current portfolio value, economic conditions) and decides to take a particular action a (e.g. the amount of shares of a particular stock to buy or sell). Following this action a, the agent receives a corresponding reward g (e.g. return on current portfolio, utility of current wealth) and transitions to a new state. Rewards can be any real numbers and will indicate whether a specific action has led to a relatively good or bad outcome. The goal is to learn what actions maximized the total reward over time. In other words, we want to train the agent in a way that when it observes any state s, it could tell us what is the best way of acting in that state.

1.2 Environment

Financial decision making and RL problems involve uncertainty and stochasticity which leads us to model the environment as a stochastic Markov Decision Process (MDP). For a discrete time MDP with $t = 0, 1, ..., T$, the environment at each time-step can be characterized by states $s_t \in \mathcal{S}$, actions $a_t \in \mathcal{A}(s_t)$ and rewards $g_{t+1} \in \mathcal{G}$. The next state s_{t+1} is a random variable with probability density determined by the current state-action pair (s_t, a_t). When the state space $\mathcal{S}$ is discrete (i.e., states can take a finite number of values), the probability transition function is:

$$p(s_{t+1} = s' | s_t, a_t)$$

This represents the probability of moving to state s' given that the agent was in state s_t and selected action a_t. When the state space $\mathcal{S}$ is continuous, the conditional probability of s_{t+1} being in some

region $\mathbb{S}_{t+1}$ given s_t, a_t is:

$$P(s_{t+1} \in \mathbb{S}_{t+1}|s_t, a_t) = \int_{\mathbb{S}_{t+1}} p(s'|s_t, a_t)ds'$$

Another important variable to consider is the reward g_{t+1} that depend on the state-action trajectory followed. Because transitions are stochastic, the reward is received after moving to state s_{t+1} which makes g_{t+1} a result of transition (s_t, a_t, s_{t+1}). Thus, the one-step reward can be defined by a function of (s_t, a_t, s_{t+1}) as:

$$g_{t+1} = g(s_t, a_t, s_{t+1})$$

Consequently, the dynamics of the environment is governed by the transition probabilities $p(\cdot|s_t = s, a_t = a)$ and the reward g_{t+1}. Indeed, such an environment describes well the dynamic decisions associated with stock trading and portfolio allocation in a stochastic stock market in which prices are generally given. Portfolio managers have to choose the stocks or assets in which to invest as well as the proportions of each in the overall portfolio.

1.3 Policy

The term **policy** is used to define the way according to which an agent chooses actions in a given state. Mathematically, a policy $\pi(a_t|s_t)$ is a function $\pi : \mathcal{S} \to \mathcal{A}(s)$ that maps states to actions. Policies are divided into two types:

- **Deterministic**: The policy is simply the action that is selected in a specific state: $\pi(a_t|s_t) = a_t$. This means that only one discrete action can be chosen in every state. In other words, given the current state s_t, we select a specific action with probability 1 while all other actions have probabilities of 0 of being selected.
- **Stochastic**: The policy represents a function that maps states s_t to probabilities of taking an action a_t with $\pi(a_t|s_t) \in (0, 1)$. In other words, the policy represents a probability distribution over actions so that at any specific state s_t, different actions can be selected at each time. Clearly, actions having higher values of $\pi(a_t|s_t)$ are more likely to be chosen at state s_t.

1.4 Rewards and Objective

Reinforcement learning is in essence a large optimization problem where the agent must find an optimal policy π (select best actions) that will maximize its expected total reward over the entire trajectory. For a finite-horizon problem, we define a full state-action trajectory of fixed, finite length T (the horizon) noted $\tau = (a_0, s_0, a_1, s_1..., s_{T-1}, a_{T-1}, s_T)$. In some cases, it is useful to note by $\tau_{t,t+h}$ a trajectory of length h starting at t and ending at $t + h$.

Finite-horizon (discounted) trajectory reward is obtained by accumulating one-step rewards over time. Intuitively, the total reward over trajectory $\tau_{t,T}$, discounted to time t, is defined as:

$$G(\tau_{t,T}) = \sum_{k=t}^{T-1} \gamma^{k-t} g_{k+1} = g_{t+1} + \gamma G(\tau_{t+1,T})$$

Here, $\gamma \in [0, 1]$ is the discount rate that is used to determine the importance of future rewards today or alternatively, quantify the uncertainty of future rewards.

Since the future is uncertain and rewards depend on the step-action trajectory followed, which in turn is determined by the policy π used, the goal of an RL agent is to maximize the *expected* total reward over the entire trajectory $\tau \sim \pi$. Precisely, the objective to maximize is defined as:

$$J(\pi) = \mathbb{E}_\pi \left[G(\tau) \right] = \mathbb{E}_\pi \left[\sum_{t=0}^{T-1} \gamma^t g_{t+1} \right] = \sum_{t=0}^{T-1} \gamma^t \mathbb{E}_\pi \left[g_{t+1} \right]$$

The expectation is taken over trajectory τ that is determined by policy π. Due to the Markov property, the probability of this trajectory, given a starting state $s_0 \sim p_0$, is defined as:

$$\begin{aligned} P_\pi(\tau) &= p_0(s_0) p_\pi(a_0, s_1, a_1, ..., s_{T-1}, a_{T-1}, s_T) \\ &= p_0(s_0) \prod_{t=0}^{T-1} p_\pi(s_{t+1}, a_t | s_t) \\ &= p_0(s_0) \prod_{t=0}^{T-1} p(s_{t+1} | s_t, a_t) \pi(a_t | s_t) \end{aligned}$$

Using the above probability, we can derive a useful expression for the expectation of one-step reward under policy π:

$$\begin{aligned} \mathbb{E}_\pi \left[g_{t+1} \right] &= \mathbb{E}_\pi \left[\mathbb{E}[g_{t+1} | s_t, a_t] \right] \\ &= \sum_\tau \mathbb{E}[g_{t+1} | s_t, a_t] P_\pi(\tau) \\ &= \sum_{\tau_{0,t}} \mathbb{E}[g_{t+1} | s_t, a_t] P_\pi(\tau_{0,t}) \\ &= \sum_{s_0, a_0} \cdots \sum_{s_t, a_t} \mathbb{E}[g_{t+1} | s_t, a_t] \left(p_0(s_0) \prod_{k=0}^{t} p(s_{k+1} | s_k, a_k) \pi(a_k | s_k) \right) \end{aligned}$$

Note that in the above equation, passing from the sum over τ to the sum over $\tau_{0,t}$ is possible due to the fact that the reward g_{t+1} is not determined by states and actions that are observed after time $t+1$ but only by what was observed up until $t+1$.

1.5 Value functions

Having defined the total reward, it is important to introduce two types of functions that are crucial in the implementation of all RL algorithms: the state value functions (V-functions) and the state-action value functions (Q-functions).

The **V-function** of state s_t under policy π, noted $V^\pi(s_t)$, represents the total expected reward obtained by starting from a specific state s_t and following π thereafter. The state value function is mathematically defined as:

$$V^\pi(s_t) = \mathbb{E}_\pi[G(\tau_{t,T}) | s_t] = \mathbb{E}_\pi[g_{t+1} + \gamma G(\tau_{t+1,T}) | s_t]$$

The **Q-function** under policy π, at a state s_t and action a_t, noted $Q^\pi(s_t, a_t)$, represents the total expected reward obtained when starting at a specific state s_t, selecting a given action a_t and

applying π thereafter. The state-action value function can be seen as to define the benefit obtained from choosing action a_t in a particular state s_t. By definition, the Q-function is equal to:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{\pi}[G(\tau_{t,T})|s_t, a_t] = \mathbb{E}_{\pi}[g_{t+1} + \gamma G(\tau_{t+1,T})|s_t, a_t]$$

1.6 Bellman Equation

The Bellman Equation is a key concept of dynamic programming that is particularly used not only in finance, but also in RL. It gives a relationship between the value of a current state and the values of its successor states. Considering the previous equation for the V-function under policy π, we can derive the associated Bellman equation:

$$\begin{aligned} V^{\pi}(s_t) &= \mathbb{E}_{\pi}[g_{t+1} + \gamma G(\tau_{t+1,T})|s_t] \\ &= \mathbb{E}_{\pi}\left[g_{t+1} + \gamma \mathbb{E}_{\pi}[G(\tau_{t+1,T})|s_{t+1}]\big|s_t\right] \qquad \text{by the law of iterated expectations} \\ &= \mathbb{E}_{\pi}[g_{t+1} + \gamma V^{\pi}(s_{t+1})|s_t] \end{aligned}$$

Similarly, from the definition of the Q-function, the corresponding Bellman equation is:

$$\begin{aligned} Q^{\pi}(s_t, a_t) &= \mathbb{E}_{\pi}[g_{t+1} + \gamma G(\tau_{t+1,T})|s_t, a_t] \\ &= \mathbb{E}_{\pi}\left[g_{t+1} + \gamma \mathbb{E}_{\pi}[G(\tau_{t+1,T})|s_{t+1}, a_{t+1}]\big|s_t, a_t\right] \\ &= \mathbb{E}_{\pi}[g_{t+1} + \gamma Q^{\pi}(s_{t+1}, a_{t+1})|s_t, a_t] \end{aligned}$$

1.7 Optimization

Recall that the goal of the RL agent is to find an optimal policy π that maximizes $J(\pi)$. If a policy π^* is better than or equal to any other policy π, then we necessarily have $V^{\pi^*}(s_t) \geq V^{\pi}(s_t)$ for all states $s_t \in \mathcal{S}$. This brings us to define below the *Bellman Optimality Equation* associated with optimal V-function, V^*, and optimal Q-function, Q^*.

$$\begin{aligned} V^*(s_t) &= \max_{\pi} V^{\pi}(s_t) \\ &= \max_{a_t} Q^*(s_t, a_t) \\ &= \max_{a_t} \sum_{s_{t+1}} p(s_{t+1}|s_t, a_t)\left(g_{t+1} + \gamma V^*(s_{t+1})\right) \\ Q^*(s_t, a_t) &= \max_{\pi} Q^{\pi}(s_t, a_t) \\ &= \mathbb{E}_{\pi}[g_{t+1} + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})|s_t, a_t] \\ &= \sum_{s_{t+1}} p(s_{t+1}|s_t, a_t)\left(g_{t+1} + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})\right) \end{aligned}$$

The optimal policy, π^*, is thus found from either the optimal V-function or from the optimal Q-function. This is done by recursively computing, at each time-step, an optimal action that

maximizes V^π or Q^π. In other words, to apply the Bellman Optimality Equation, at each step t, we must map and memorize the state, action, and the corresponding value of the $V-$ or $Q-$ function. Depending on the action-state spaces, there could be a rapid increase of computation and memory storage. In addition, it should be noted that finding π^* from the V-function can be harder since it involves taking the maximum over an expectation, which can be difficult in uncertain environments with an unknown MDP.

1.7.1 Approximations

For most financial tasks, we either cannot find an accurate model to describe the environment, or there are too many states and/or actions to visit and keep track of. When the dimensions of the state-and-action space increase significantly, storing all state, action, and optimal value function combinations in memory is not efficient. Deep Reinforcement Learning (DRL) deals with such issues by using Deep Neural Networks (DNNs)[1] as function approximators to estimate the optimal value functions and/or the optimal policy distribution. In other words, a neural network will map states and actions to optimal value functions and could also map states to optimal actions.

We start by an approximation of *optimal* value functions. Let the V-function and the Q-function be parameterized by ν and ω, respectively, giving corresponding DNNs $V_\nu(s_t)$ and $Q_\omega(s_t, a_t)$. The V-function network will take the vector of states (or states' features) as input and will output a single value corresponding to $V^*(s_t)$ for every state. The Q-function network's input corresponds to the state-action pair (s_t, a_t) and outputs a value of $Q^*(s_t, a_t)$ corresponding to each state-action tuple. Consequently, the sub-optimal policy $\hat{\pi}^*(a_t|s_t)$ can be found using the estimated optimal value functions such as:

$$\hat{\pi}^*(a_t|s_t) \in \arg\max_{a_t} \mathbb{E}_\pi[g_{t+1} + \gamma V_\nu(s_{t+1})|s_t, a_t]$$

or

$$\hat{\pi}^*(a_t|s_t) \in \arg\max_{a_t} Q_\omega(s_t, a_t)$$

When the exact model of the environment is not known, the computation of expectation is done through Monte Carlo simulation. Moreover, since the true optimal functions are not known, the loss and the parameters updates are computed using some target function $y(\cdot)$. The definition of the target function depends on the RL algorithm used but often, we use one of the following:

$$y(s_{t+1}) = g_{t+1} + \gamma V_\nu(s_{t+1})$$

or

$$y(s_{t+1}, a_{t+1}) = g_{t+1} + \gamma \max_{a_{t+1}} Q_\omega(s_{t+1}, a_{t+1})$$

In general, at each iteration of the algorithm, parameters of the value function are updated by minimizing the squared loss between the target, $y(\cdot)$, and an estimation of the value function, V_ν or Q_ω.

Having defined a general procedure for value function estimation, we proceed to approximations in policy space. We assume that the optimal policy π is approximated by a neural network π_θ, and

[1] Refer to Appendix B for DNN definitions and descriptive features.

parameterized by θ. When dealing with stochastic policies, the policy is frequently chosen to be from the Gaussian family of distributions such that:

$$\pi_\theta(a_t|s_t) = \frac{1}{\sqrt{2\pi}\sigma(s_t,\theta_\sigma)} \exp\left(\frac{-(a_t - \mu(s_t,\theta_\mu))}{2\sigma(s_t,\theta_\sigma)^2}\right)$$

In other words, an action a_t is selected by sampling from the normal distribution defined by neural networks $\mu(s_t,\theta_\mu)$ and $\sigma(s_t,\theta_\sigma)$ that approximate, respectively, the mean and the standard deviation of a gaussian. The standard deviation must always be positive and thus, is typically defined as an exponential of some function. Note that if any other form of distribution is chosen, the parameters of that distribution can also be approximated by neural networks.

To understand how to find optimal parameters θ of π_θ, we must explain the Policy Gradient Theorem.

1.7.2 Policy Gradient Theorem

Policy Gradient aims to maximize the objective $J(\pi_\theta)$ given that the sequence of optimal actions follows policy π_θ. We begin by setting the gradient of $J(\pi_\theta)$ to zero and use the definition of $J(\cdot)$ derived in section 1.4. to obtain the following expression[2]:

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{s=1}^{T-1} \nabla_\theta \log \pi_\theta(a_s|s_s) Q^{\pi_\theta}(s_s,a_s)\right] \\ &\sim \frac{1}{|\mathcal{D}|}\sum_{\tau\in\mathcal{D}}\sum_{s=1}^{T-1} \nabla_\theta \log \pi_\theta(a_s|s_s) Q^{\pi_\theta}(s_s,a_s)\end{aligned}$$

The last expression is a MC estimate of the policy gradient with $\mathcal{D}$ being a set of trajectories. Intuitively, a step in the policy gradient direction should increase the probability of better-than-average actions and decrease the probability of worse-than-average actions. Assuming that we are able to calculate $\nabla_\theta \log \pi_\theta(a_t|s_t)$, we can compute the policy gradient and update parameters using

$$\theta_{k+1} \leftarrow \theta_k + \alpha \nabla_{\theta_k} J(\pi_{\theta_k})$$

2 Deep Reinforcement Learning Algorithms

When the transition probabilities associated with the MDP are unknown, we must estimate the expectation term involved in the definition of the value functions. Moreover, although analytical solutions exist for some problems involving continuous action and/or state spaces (e.g., maximization of quadratic expected utility), the maximization problem rarely has an analytical solution and thus, it becomes hard to find the maximum of value functions and compute the expectation. To deal with this problem, Monte Carlo (MC) methods can be used in tandem with value functions and policy approximations. The approximation is either done by using neural networks (for continuous action/spaces) or by using linear approximations (when state-action spaces can be discretized). This makes us introduce some model-free RL algorithms such as Proximal Policy Optimization (PPO), Advantage Actor-Critic (A2C/A3C), and Soft Actor-Critic (SAC). These algorithms use

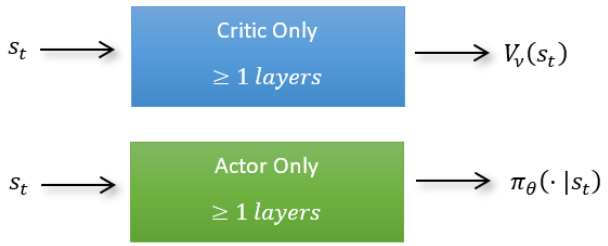
[2]See Appendix A for the derivation of the Policy Gradient Theorem expression.

experience to sample state-action trajectories from actual or simulated interaction with an environment. In addition, they use neural networks to estimate the optimal value function and the optimal policy to follow.

2.1 Actor Critic Methods

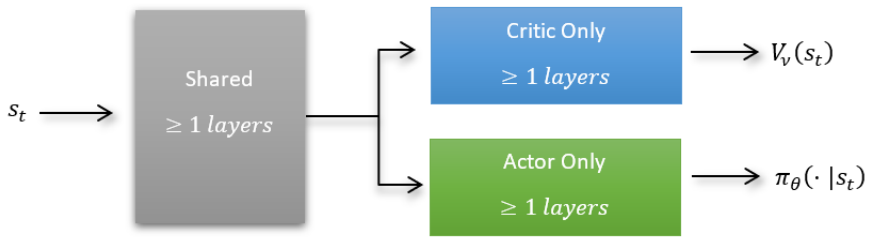
Actor and Critic (AC) algorithms divide the agent into two parts: an actor (policy) and a critic (value). These algorithms make two approximations using neural networks. The "Critic" network estimates the value of a state and possibly evaluates how good is the chosen action. This could be an estimate of the V-function or the Q-function. The "Actor" updates the policy distribution in the direction suggested by the Critic, it is based on the policy gradient approach. Generally, separate neural networks architectures are used to approximate the Critic and the Actor. This means that we must define two networks and learn two different functions. The structure of a separate network is illustrated in the following graph:

Separate Network Architecture



Alternatively, one network that will output both, the optimal value function and the optimal policy distribution, can be used. The latter network is referred to as shared network in Deep Learning terminology and has the following form:

Shared Network Architecture



When a state s_t is passed to the shared network, it will pass through at least one shared layer. The output of the last shared layer will be further split into two separate layers to obtain two separate "sub-networks".

2.1.1 Advantage Actor-Critic

The AC algorithm that we will focus on is called the Advantage Actor-Critic, or A2C for short. As its name suggests, the A2C algorithm is based on an **advantage function**. To understand the advantage function, recall that in Policy Gradient methods, we are interested to know what is the

benefit of taking a certain action in a given state. The satisfaction received from a certain action depends on the environment's state, that is, one "good" action will result in a greater benefit if the agent is in a state that is closer to the target than if the agent is far from achieving that target. Hence, it is useful to determine the relative advantage of a particular action at a given state.

The advantage function $A^\pi(s, a)$ corresponding to policy π describes how much value is added or subtracted when given a specific state s, we decide to take a specific action a according to policy π. That is, it is the difference between the state-action value function (Q-function) and the state value function (V-function):

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$$

It can be seen that the advantage at time t will be negative if we are in a favorable state (high V^π) but we choose a "bad" action (low Q^π) Typically, the first step is to obtain an estimate of the value of a state, which is the expected reward the agent will receive, given that it already arrived at the present state. Second, the action-value function must be computed. The action-value function represents the value of a state and of a chosen action *after* that action was made. In our context, the approximations are obtained with neural networks V_ν and Q_ω. However, it is often impossible or complicated to calculate the Q-function so a good estimate of $Q^\pi(s_t, a_t)$ will be the reward received from performing an action a_t at a given state s_t defined as $g_{t+1} + \gamma V^\pi(s_{t+1})$. The latter gives rise to the the unbiased estimator of the advantage function, defined as the Temporal Difference (TD) error:

$$\delta_t^\pi = g_{t+1} + V^\pi(s_{t+1}) - V^\pi(s_t)$$

When we use DNNs to estimate the Q-function and the V-function, the *hat* notation will be used to indicate that the advantage and the TD error are computed from value function approximations:

$$\hat{A}(s_t, a_t) = Q_\omega(s_t, a_t) - V_\nu(s_t)$$

$$\hat{\delta}_t = g_{t+1} + \gamma V_\nu(s_{t+1}) - V_\nu(s_t)$$

The above definitions are very useful for the policy gradient methods because as trajectories during training can significantly deviate from each other, it is suitable to reduce this variability by subtracting some baseline function $b(s_t)$ from the Q-function or from cumulative rewards $G(\tau_{s,T})$ to make these quantities smaller. When the Q-function or cumulative rewards are reduced by $b(s_t)$, gradient' values decrease which in turn, stabilizes updates. In fact, we can generalize the policy gradient expression to:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=1}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \Phi_t \right],$$

where Φ_t can be defined as one of the following:

- Total reward of trajectory: $G(\tau)$
- Reward to-go after taking an action at step t: $G(\tau_{t,T})$
- Baselined version using rewards to-go: $G(\tau_{s,T}) - b(s_t)$
- On policy Q-function: $Q^\pi(s_t, a_t)$
- Advantage function $A^\pi(s_t, a_t)$

- TD error δ^{π}

The **A2C** algorithm uses an estimated advantage function $\hat{A}(\cdot)$ to update parameters. The Actor is given by the policy network π_θ, parameterized by θ, and the Critic is given by the state value function network V_ν, parameterized by ν. For a finite-horizon process, ending at time T, the idea is to sample state-action transitions using the current estimate of the policy, π_θ, compute the targets of the V-function and the corresponding advantage, and finally, update the parameters using a regression for the value function and gradient ascent for the policy.

The most basic A2C algorithm uses the one-step look-ahead target y_t. This target is derived from the Bellman Equation for the state value function and is given by $g_{t+1} + V_\nu(s_{t+1})$. This allows us to approximate the advantage by the one-step TD error $\hat{\delta}_t$ which is computed using the approximator of the V-function. Parameters of the Critic are updated by minimizing the following loss function:

$$L(\nu) = (y_t - V_\nu(s_t))^2 = (\hat{\delta}_t)^2$$

Parameters of the Actor are updated using the policy gradient theorem where the actor loss is:

$$L(\theta) = -\log \pi_\theta(a_t|s_t)(y_t - V_\nu(s_t)) = -\log \pi_\theta(a_t|s_t)\hat{\delta}_t$$

At each time step, the one step look-ahead A2C algorithm executes one action a_t, moves to next state s_{t+1}, receives one reward g_{t+1} and makes a training update based on that one transition. As a result, it constantly learns with every step. The downside of this approach is that it cannot benefit from larger batch sizes (that has better computational efficiency and a stabilizing effect), and the sequential training data examples in one episode are highly correlated, which is problematic for training higher dimensional RL problems.
A more generalized form of the A2C, that stabilizes training by increasing batch sizes, is the one that uses a n-step look-ahead update instead of a one-step update. In other words, we look n steps into the future and compute the n-step total reward $G(\tau_{t,t+n})$. In addition, instead of the one-step TD error, we use the n-step TD error defined as[3]:

$$\hat{\delta}_t^{(n)} = \sum_{k=t}^{t+n-1} \gamma^{k-t} g_{k+1} + \gamma^n V_\nu(s_{t+n}) - V_\nu(s_t)$$

Note that if $t+n \geq T$, then we must set $V_\nu(s_{t+n}) = 0$ since at $t+n$, the horizon has already reached its terminal state. The update is done by using the previously defined losses $L(\nu)$ and $L(\theta)$. The detailed pseudo-algorithm for a n-step A2C is the following:

[3]See Appendix A for the equation derivation

Algorithm 1 A2C

Initialize the actor and the critic parameters θ and ν
Select learning rates $0 < \lambda_\theta,\ \lambda_\nu \leq 1$
Select the discount rate $0 \leq \gamma \leq 1$
for epoch $0, 1, ...N$
    Reset environment, set the number of steps for update $= n$
    Generate trajectory $\tau = (s_0, a_0, ..., a_{T-1}, s_T) \sim \pi_\theta$
    Calculate corresponding rewards over trajectory $g_{t+1} = g(s_t, a_t, s_{t+1}) \forall t = 0, ..., T-1$
    for $t = T-1$ **to** 0 :
        Define done flag at step k: $d_k = \mathbb{I}\{k \geq T\}$
        Calculate the n-step reward using Critic

$$G(\tau_{t,t+n}) = \sum_{k=t}^{t+n-1} \gamma^{k-t} g_{k+1}(1-d_k) + \gamma^n (1-d_{t+n}) V_\nu(s_{t+n})$$

        Compute the n-step TD error $\hat{\delta}_t^{(n)} = G(\tau_{t,t+n}) - V_\nu(s_t)$
    end for

    Compute losses

$$L(\theta) = -\frac{1}{T} \sum_{t=0}^{T-1} \log \pi_\theta(a_t|s_t) \hat{\delta}_t^{(n)}$$
$$L(\nu) = \frac{1}{T} \sum_{t=0}^{T-1} \left(\hat{\delta}_t^{(n)}\right)^2$$

    Update networks' parameters:

$$\theta = \theta + \lambda_\theta \nabla_\theta L(\theta)$$
$$\nu = \nu + \lambda_\nu \nabla_\nu L(\nu)$$

end for

In the above pseudo-algorithm, epochs form the outer loop of the training algorithm and there is one value and policy update per epoch. Typically, the total number of epochs, N, is set to be significantly large in order to reach a good solution. Depending on the problem, a specific discount rate must be chosen but a popular choice is to select it within a range of $\gamma = 0.9$ and $\gamma = 0.99$. Learning rates are hyperparameters (chosen by the user) that will control the degree of exploration. Precisely, a small learning rate will make smaller gradient updates and require more training epochs while large learning rates will significantly change the parameters in one gradient step thus, requiring less training epochs. Note that if the learning rate is too small, the algorithm might never find an optimal solution while if it is too large, the algorithm might diverge. Consequently, one of the challenges of training neural networks is to select a good learning rate.

Alternatively, if a shared network is used, instead of computing two separate losses and updating

separate parameters, we compute one loss function:

$$L(\theta_{\pi,v}) = L(\theta) + c_v L(\nu) = \frac{1}{T}\sum_{t=0}^{T-1}\Big(-\log \pi_\theta(a_t|s_t)\hat{\delta}_t^{(n)} + c_v(\hat{\delta}_t^{(n)})^2\Big),$$

where $\theta_{\pi,v}$ represents the shared parameters vector for both, the Actor and the Critic, and constant $c_v \in \mathbb{R}^+$ is the hyperparameter that controls the importance of value function updates. The coefficient c_v is typically set < 1 to reduce the contribution from the value function loss.

2.1.2 Entropy Regularization

At any given point in time, the knowledge of an agent about the state, actions, rewards and resulting states is partial which results in the so called *Exploration-Exploitation Dilemma*. Suppose you have some partial information about the environment and the value of some, but not all, actions. **Exploration** allows the agent to improve its knowledge about each action which could lead to a higher future total reward. **Exploitation**, on the other hand, only uses the current knowledge and exploits the agent's current estimated value in choosing optimal actions. However, in the latter case, the agent has only partial knowledge and does not know the actual value of its actions, so chances are it might not get the most reward. The balance between exploration and exploitation can be described with the degree of uncertainty for action selection.

Recall that the optimal policy, $\pi(a_t|s_t)$, is represented by the probability distribution of actions, a_t, for each state s_t. This probability represents the uncertainty of the action selection and the learning process is essentially the process of reducing this uncertainty of which action to choose at each specific state. At the beginning, we would like to explore the environment the most implying that each action is equally likely to be chosen or, equivalently, that the uncertainty is maximized. Therefore, we introduce a new notion of *entropy* to measure the degree of uncertainty and serve as an exploration tool. In our context, the entropy measures the uncertainty of the policy distribution $\pi(a_t|s_t)$ and is typically defined as:

$$H_\pi(s_t) = \mathbb{E}[-\log \pi(a_t|s_t)]$$

The more uncertain a policy's distribution is, the more diverse actions it produces and the higher is the value of $H_\pi(s_t)$. Conversely, less uniform policies produce similar actions and have lower entropy.

The entropy term is subtracted from the policy loss function, defined in the previous section, with its corresponding coefficient $c_h \geq 0$ so that the policy loss becomes:

$$L(\theta) = \frac{1}{T}\sum_{t=0}^{T-1}\Big(-\log \pi_\theta(a_t|s_t)\hat{\delta}_t^{(n)} - c_h H_\pi(s_t)\Big)$$

Because the entropy measure H_π and the hyperparameter c_h are always non-negative, the term $-c_h H_\pi(s_t)$ is always negative. When a policy is certain, entropy decreases implying that $-c_h H_\pi(s_t)$ increases and contributes more to the loss to encourage more exploration. Conversely, when a policy is very uncertain, entropy increases and $-c_h H_\pi(s_t)$ decreases and contributes less to the loss, which makes the policy to have little incentive to change through the entropy term.

2.2 Proximal Policy Optimization (PPO) - Clip Version

[4] The second algorithm we present is the Proximal Policy Optimization (PPO) algorithm that uses clipped surrogate objective. PPO is a policy optimization method that indirectly optimizes θ by maximizing local approximations of the objective $J(\theta)$. The main ideas presented for A2C remain the same for PPO, that is, both algorithms have an actor and a critic improving each other in the learning process. Moreover, both can be used for discrete and continuous action and state spaces. However, with A2C, the magnitude of a policy update might drastically increase with an action's increasing probability. In other words, a newly discovered action can be mistakenly viewed by the actor as highly gainful, making the policy bad at exploration as the agent continues to choose the unfavorable action over and over again. This cycle of exploitation causes a significant shift in policy parameters towards that action and with every update, losses become exceedingly large. So even if the agent performed well at the beginning, a slight change in the policy parameters can cause a high drop in performance, without a possibility for further recovery.

PPO helps to fix this instability issue by restricting large parameters updates so that multiple steps of optimization can be done without actor's action distribution fluctuating too much. Precisely, the algorithm ensures that the loss cannot get higher than the allowed ***clip-ratio***, noted by ϵ. First, instead of using the log-probability of an action as in the A2C policy update, PPO uses the following ***policy ratio*** to trace the impact of the actions:

$$h_t(\theta, \theta_{old}) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

By definition, $h_t(\theta, \theta_{old})$ is the ratio between the probability of the action under current policy parameters, represented by vector θ, and the probability of the action under previous parameters, represented by vector θ_{old}. When the action is more likely under the current policy than it is under the old policy, this ratio will be greater than 1 and vice-versa.

To limit the size of the policy update, we will also use the clipped policy ratio:

$$\overline{h}_t(\theta, \theta_{old}; \epsilon) = \text{clip}\left(h_t(\theta, \theta_{old}), 1-\epsilon, 1+\epsilon\right)$$

In the above expression, the clipped policy ratio, $\overline{h}_t(\theta, \theta_{old}; \epsilon)$, introduces lower and upper bounds on the policy ratio so that $h_t(\theta, \theta_{old})$ stays within the interval $(1-\epsilon, 1+\epsilon)$. Consequently, PPO only considers rewards that are received from state-action pairs that satisfy $1-\epsilon < h_t(\theta, \theta_{old}) < 1+\epsilon$. Note that the clip ratio is a chosen hyperparameter that corresponds to by how much the new policy can deviate from the old one. It is now possible to derive an expression for the PPO objective to ***maximize***:

$$L^{clip}(\theta) = \min\left(h_t(\theta, \theta_{old})\hat{A}^{\theta_{old}}(s_t, a_t), \quad \overline{h}_t(\theta, \theta_{old}; \epsilon)\hat{A}^{\theta_{old}}(s_t, a_t)\right)$$

The maximization of $L^{clip}(\theta)$ is done with respect to θ while the advantage is calculated based on the previously obtained parameters θ_{old}. If the action was estimated to be favorable (when $\hat{A} > 0$) and it became a lot more likely under the current policy than it was prior to the update (high

[4] Original https://arxiv.org/pdf/1707.06347.pdf
Improvements https://arxiv.org/pdf/2009.10897.pdf
Paper on A2C and PPO https://arxiv.org/pdf/2009.10897.pdf

values of $h_t(\theta, \theta_{old})$), the policy ratio gets clipped and the gradient step does not get too large. On the other hand, if the action was estimated to be unfavorable (when $\hat{A} < 0$) and it became a lot less likely under the current policy, we do not want to continue making it less and less likely and thus, activate the clipping again.

Finally, for a trajectory of length T generated by π_θ over all time steps, the approximate advantage function is computed as:

$$\begin{aligned}\hat{A}^\theta(s_t, a_t) &= g_{t+1} + \gamma g_{t+2} + ... + \gamma^{T-t-1} g_T + \gamma V_\nu(s_T) - V_\nu(s_t) \\ &= \hat{\delta}_t + \gamma\hat{\delta}_{t+1} + ... + \gamma^{T-t-1}\hat{\delta}_{T-1}\end{aligned}$$

The detailed PPO algorithm is the following:

Algorithm 2 PPO-Clip, One Agent, Separate Networks

Initialize the actor and the critic parameters θ and ν
Select value learning rate $0 < \lambda_\nu \leq 1$
Select the discount rate $0 \leq \gamma \leq 1$
Select the clip-ratio value, a good choice is $\epsilon = 0.2$
for epoch $0, 1, ...N$
    Reset environment
    Generate trajectory $\tau = (s_0, a_0, ..., a_{T-1}, s_T) \sim \pi_\theta$
    Calculate corresponding step rewards $g_{t+1} = g(s_t, a_t, s_{t+1}) \forall t = 0, ..., T-1$

    for $t = T-1$ **to** 0 :
        Compute rewards-to-go $G(\tau_{t,T})$ using critic
        Compute advantage estimate $\hat{A}^{\theta_{old}}(s_t, a_t)$
    end for

    Randomly sample a batch of available data on a fixed time horizon of length $T_M \leq T$
    Update policy parameters:

$$\theta = \arg\max_\theta \frac{1}{T_M} \sum_{t=0}^{T_M-1} \min\Big(h_t(\theta, \theta_{old})\hat{A}^{\theta_{old}}(s_t, a_t), \quad \overline{h}_t(\theta, \theta_{old}; \epsilon)\hat{A}^{\theta_{old}}(s_t, a_t)\Big)$$

$$\theta_{old} \longleftarrow \theta$$

    Compute value mean loss:

$$L(\nu) = \frac{1}{T_M} \sum_{t=0}^{T_M-1} \Big(G(\tau_{t,T_M}) - V_\nu(s_t)\Big)^2$$

    Update the value network parameters:

$$\nu = \nu + \lambda_\nu \nabla_\nu L(\nu)$$

end for

Such an approach prevents the learning from being sensitive to minor perturbations and thus,

stabilizes the overall process. Note that when the parameters of actor and critic networks are shared (one network is used for both), only the parameter vector θ is updated using the following objective function:

$$L^{clip-shared}(\theta) = L_{clip}(\theta) - c_v \left(G(\tau_{t,T} - V_{\theta_{old}}(s_t))\right)^2 + c_h H_{\pi_\theta}(s_t)$$

Where $L^{clip}(\theta)$ is the clipped surrogate objective, $H_{\pi_\theta}(s_t)$ is the entropy of policy π_θ with entropy coefficient c_h, and c_v is the value loss function coefficient.

3 Applications in Finance

To illustrate the functioning of the DRL algorithms, we consider a portfolio allocation problem whereby an investor has to choose how much to invest in a number of stocks in a dynamic fashion in order to maximize her wealth over a certain time period. The training process starts by observing stocks' key features such as returns, correlations of returns between stocks, technical indicators, and other stock characteristics. Further, it involves taking an action (choosing an allocation) and computing corresponding rewards so that the agent adjusts its strategy accordingly. As a result, by interacting with the environment, the agent will derive an optimal strategy with the maximized total reward. In this section we define multiple variants of a portfolio allocation problem and describe how they can be solved using the Advantage Actor-Critic algorithm.

3.1 Portfolio Allocation Problems

General multi-period portfolio selection problems can be summarized as distributing available capital among different assets (such as bonds, stocks, cryptocurrencies) over a given time horizon as to achieve a specific objective. To formulate the portfolio allocation task as a finite-horizon MDP, in which an agent must make decisions at discrete time steps over its investment horizon, it is necessary to define the corresponding environment. For our applications, we use the following assumptions and notations to characterize the finite-horizon framework:

- The investor makes its allocation decisions during an investment horizon of $T < \infty$ discrete periods, with each period indexed by $t = 0, 1, ..., T$.
- There are no taxes but there may be transaction costs.
- The portfolio consists of N stocks. Stock j's closing price is available and is noted $p_t^{(j)}$. The vector of all stocks' closing prices is $p_t \in \mathbb{R}^{N\times 1}$ for $t = 0, 1, ..., T-1$.
- Stock returns can be found from prices such as the simple return of asset j for period t is calculated as $r_t^{(j)} = p_{t+1}^{(j)}/p_t^{(j)} - 1$. The vector of asset returns is noted $r_t \in \mathbb{R}^{N\times 1}$ for periods $t = 0, 1, ..., T-1$.
- At time $t = 0$, the investor has an initial capital of $W_0 > 0$ dollars available for investment.
- Each trading period, the agent observes the stock market by analyzing data and then reallocates its portfolio. Precisely, the investor observes a state vector/matrix $s_t = (s_t^{(1)}, ..., s_t^{(N)})$ where each $s_t^{(j)}$ is a scalar or vector containing the closing price of stock j at time t and may include additional information such as the volume of shares of stock j, correlation of stock

j with other stocks in the portfolio, technical indicators such as exponential moving average (EMA) or average directional index (ADX), macroeconomic factors, and more.

- Given state s_t, the investor must choose an action from an action space $\mathcal{A}(s_t)$ which contains values of the allowed actions. The action space may be either discrete or continuous. In the discrete case, the investor decides on the *number of shares* to trade (buy or sell) such that the decision for asset j at time t is $a_t^{(j)} \in \{-k_j, ..., 0, ..., k_j\}$ where k_j represents the maximum number of shares of stock j that could be bought or sold. In the continuous case, the decision vector represents target weights in each asset such that $a_t \in [0, 1]^N$ for a long-only portfolio and $a_t \in [-1, 1]^N$ if short sales are allowed.

- Based on the chosen action a_t, the agent moves to the next time-step, observes next state s_{t+1} and receives a numerical reward $g_{t+1} = g(s_t, a_t, s_{t+1})$.

The form of the reward function g_{t+1} is mainly determined by the objective the investor aims to achieve. Examples of common objectives include optimization of profit/return, minimization of risk and optimization of economic utility. When maximizing profit of a risk-insensitive investor, the most simple and natural definitions of instantaneous reward are the one-period profit and one-period portfolio log-return. Using the price and the stock returns vectors, p_t and r_t, at time t and the action vector a_t representing the desired allocation *weights*, the new portfolio value, noted P_{t+1}, and the portfolio (simple) return from period t to $t + 1$, noted R_t, are computed as:

$$R_t = a_t \cdot r_t = \sum_{j=1}^{N} a_t^{(j)} \left(\frac{p_{t+1}^{(j)}}{p_t^{(j)}} - 1 \right)$$
$$P_{t+1} = P_t(1 + R_t) \quad \text{and} \quad P_0 = W_0$$

Alternatively, if the objective is the minimization of risk, the reward is defined in terms of portfolio variance. Since this is a minimization problem, we must add a negative sign to obtain an equivalent maximization problem for the DRL algorithms described. We can now proceed to mathematically define simple common forms of the reward function:

- New portfolio value: $g_{t+1} = P_{t+1}$
- Portfolio log-return (if actions are fractions): $g_{t+1} = \log \left(\frac{P_{t+1}}{P_t} \right)$
- Portfolio variance: $g_{t+1} = -a_t^\top \Sigma_t a_t$

Note that the covariance matrix of asset returns defined by Σ_t is estimated using the covariance of past stock returns. Past stock returns, in our setting, are obtained at each time step t by taking τ previous return observations for each stock. Having obtained a sequence of rewards g_{t+1}, we proceed to compute the total trajectory reward that will be used in the Advantage Actor-Critic algorithm.

For the next examples, each time-step t will correspond to a trading month and the risky portfolio will consist of $N = 4$ selected stocks from the Dow Jones Index (DJI).

Example 1 - Baselines

The following examples serve as baselines. In the first part, we assume the general framework defined previously where the investor is interested in risk minimization. In the second part, the investor wants to maximize its return in the presence of transaction costs.

No transaction costs

The state space of the trading environment consists of monthly closing prices of $N = 4$ DJI stocks, noted by the price vector p_t as well as the covariance matrix of stocks returns Σ_t. Therefore, the state is $s_t = [p_t \quad \Sigma_t]$ which has continuous variables and is of dimension $N + N \times N$.

The action space is also formed by continuous variables (weights) such that $a_t \in [0, 1]^N$ represents the weights of each stock in the portfolio. We choose to impose that no short positions are allowed. Therefore, we must normalize the raw weights obtained from policy π_θ in order to respect the constraint of nonnegative portfolio weights summing to one. Hence, we work with normalized actions $\bar{a}_t = [\bar{a}_t^{(1)}...\bar{a}_t^{(N)}]$ where the normalization is done using the following transformation on raw weights a_t:

$$\bar{a}_t^{(j)} = \frac{a_t^{(j)}}{\sum_{i=1}^{N} a_t^{(i)}} \quad \text{for } j = 1, ..., N; t = 0, ..., T-1$$

The portfolio is initialized to be equally weighted, that is, at period (month) $t = 0$, weights are $a_0 = \bar{a}_0 = [1/N...1/N]$

In this example, we assume an investor whose goal is to minimize the portfolio risk over time. A natural definition of instantaneous MDP reward would be the negative of the portfolio estimated variance resulting from decision $\bar{a}_{t+1}$ such that:

$$g_{t+1} = -\bar{a}_t^\top \Sigma_t \bar{a}_t$$
$$\implies G(\tau) = -\sum_{t=0}^{T-1} \bar{a}_t^\top \Sigma_t \bar{a}_t$$

where Σ_t is the covariance matrix of monthly returns estimated from daily returns observed during the past year, i.e. daily returns from period $t - 12$ to t.

With transaction costs

An alternative to the problem above is to consider an investor who wants to maximize its return in the presence of a commission fee that must be paid for each transaction. Transaction costs will be defined as a certain percentage over the transaction value with the total fee at time t, given by $\psi(u_t)$, function of trades u_t. Moreover, we assume that in addition to N DJI stocks, the investment portfolio includes a risk-free cash account earning, for simplicity, a constant interest rate of $r_f \geq 0$ per period. As before, the action vector represents target allocations with the slight modification that it is now of dimension $N+1$ such that $a_t \in [0, 1]^{N+1}$. We modify the state vector to include the price vector along with some technical indicators computed from the stock price data, namely, the Moving average convergence divergence (MACD), Bolling Bands, Relative Strength Index (RSI),

Commodity Channel Index (CCI), Directional Index (DX), and Simple Moving Average (SMA). Hence, the state variables remain continuous. At each time step t, we define the state matrix as $s_t = [p_t \quad Tech_t]$ where p_t is the vector of stocks closing prices and $Tech_t$ is the matrix of chosen technical indicators. Consequently, the state matrix at each time step, is a matrix of dimension $N \times (1 + \text{number of technical indicators})$.

At the beginning of period $t = 0$, the agent is assumed to have all of its initial capital, W_0, invested in the risk-free asset which implies deterministic initial actions $\bar{a}_0 = [1 \quad 0 \quad ... \quad 0]$. The investor then waits until the end of the period such that over period $t = 0$ it accumulates a riskless rate r_f on its total capital. The accumulated wealth becomes $W_1' = W_0(1 + r_f)$ while the adjusted weight vector stays the same at the end of the first time period, $\bar{a}_1' = [1 \quad 0 \quad ... \quad 0]$. At the same time, the investor observes stock closing prices p_0^{close} and must select an optimal allocation for the next period which is represented by vector $\bar{a}_1$. Because $\bar{a}_1$ represents the new optimal allocation, the agent must rebalance its portfolio and trade a portion $u_1 = \bar{a}_1 - \bar{a}_1'$ of its current capital. This will imply transaction costs of $\delta_j^+ \geq 0$ and $\delta_j^- \geq 0$ for, respectively, buying and selling stocks $j = 1, ..., N$. Note that we define transaction costs as a percentage of the amount traded.

It follows that at the beginning of any period $t = 1, ..., T-1$, the capital is allocated according to $\bar{a}_t = [\bar{a}_t^{(0)} \quad \bar{a}_t^{(1)} \quad ... \quad \bar{a}_t^{(N)}]$. The agent waits until the end of t to observe stocks' closing prices. This information enables us to define the vector of gross returns during period $t = 1, ..., T-1$ as:

$$1 + r_t = \left[(1 + r_f) \quad (1 + r_t^{(1)}) \quad ... \quad (1 + r_t^{(N)})\right]$$

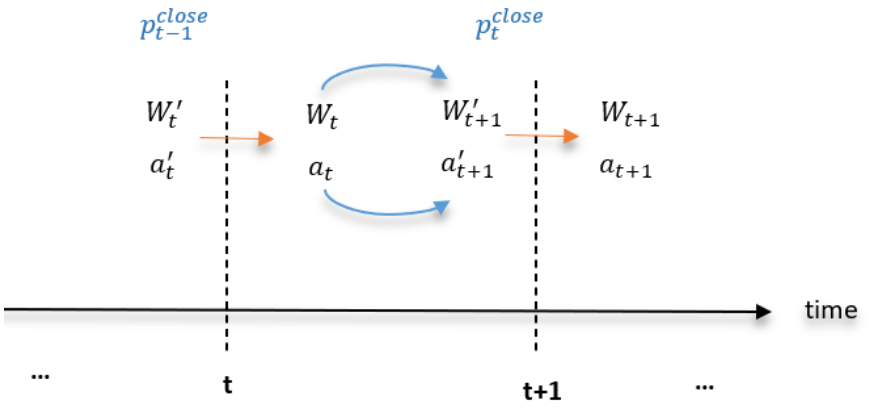
Consequently, the total wealth at the beginning of period t accumulates to $W_{t+1}' = W_t(\bar{a}_t \cdot (1 + r_t))$. Wealth evolution can also be expressed in terms of adjusted weights as follows:

$$\bar{a}_{t+1}' = \frac{(1 + r_t) \circ \bar{a}_t}{(1 + r_t) \cdot \bar{a}_t}$$

In the equation above, we have used $\circ$ to denote the element-wise product between the vector of gross returns and the vector of target allocations. After observing stock price data, the agent chooses a new allocation vector $\bar{a}_{t+1}$ and adjusts its portfolio weights by trading $u_{t+1} = \bar{a}_{t+1} - \bar{a}_{t+1}'$ fractions of current capital. If $u_{t+1}^+ \geq 0$ and $u_{t+1}^- \leq 0$ are vectors of buying and selling trades, the investor incurs total transaction costs of $\psi(u_{t+1}) = \delta^+ \cdot u_{t+1}^+ - \delta^- \cdot u_{t+1}^-$, expressed as a fraction of wealth. Wealth at the beginning of period $t+1$ is the accumulated wealth net of transaction costs,

$$W_{t+1} = W_{t+1}'(1 - \psi(u_{t+1})) = W_t(\bar{a}_t \cdot (1 + r_t))(1 - \psi(u_{t+1}))$$

For clarity, the process of wealth evolution is illustrated in the figure below.



Note that for this alternative problem definition, the vector of closing prices, p_{t-1}^{close}, is included in the state matrix at the beginning of period t. Each subsequent value of wealth can be computed as a product of the previous amount of capital and gross return, net of fees. The terminal value of wealth can thus be computed using:

$$\begin{aligned} W_T &= W_{T-1}\bar{a}_{T-1}(1+r_{T-1})(1-\psi(u_T)) \\ &= W_0 \prod_{t=0}^{T-1} \bar{a}_t \cdot (1+r_t)(1-\psi(u_{t+1})) \end{aligned}$$

with $1+r_0 = 1+r_f$ and initial allocation $\bar{a}_0$ as before. Considering transaction costs, the one-step MDP reward is re-defined as:

$$G(\tau) = \log\left(\frac{W_T}{W_0}\right) = \sum_{t=0}^{T-1} \log\left(\bar{a}_t(1+r_t)(1-\psi(u_{t+1}))\right) = \sum_{t=0}^{T-1} g_{t+1}$$

Although we have defined actions as portfolio weights, it is possible to use an alternative definition and set stock trades (change in the asset positions) as actions. Thus, we invite you to look at Appendix C for a detailed description of this alternative specification.

Example 2 - Risk Averse investors

For risk-averse investors, the degree of risk aversion varies. Hence, in addition to the risk measure, we must consider a coefficient λ that measures the degree to which an investor is concerned about taking risky decisions. In this example, our agent will want to achieve both - maximized capital and minimized portfolio risk. As before, we define the risk measure by the volatility of portfolio returns. Most of the definitions are kept from *Example 1*, and we only modify the reward function.

The state matrix includes the covariance matrix of all assets (including uncorrelated risk-free asset) and the stock price information such that $s_t = [p_t \quad Tech_t \quad \Sigma_t]$ can be placed in a matrix of dimension $N \times (\text{nb of technical indicators} + 1) + (N+1) \times (N+1)$. Recall that at the beginning of period t, the investment portfolio is constructed using allocation $\bar{a}_t$. At the end of period t, the agent observes stock price data, chooses a new optimal allocation $\bar{a}_{t+1}$, and adjusts its portfolio weights by trading $u_{t+1} = \bar{a}_{t+1} - \bar{a}'_{t+1}$ fractions of current capital. We have also defined total

transaction costs as $\psi(u_{t+1}) = \delta^+ \cdot u_{t+1}^+ - \delta^- \cdot u_{t+1}^-$, expressed as a fraction of wealth. The next step total wealth is calculated using:

$$W_{t+1} = W'_{t+1}(1 - \psi(u_{t+1})) = W_t(\bar{a}_t \cdot (1 + r_t))(1 - \psi(u_{t+1}))$$

Since we are concerned about risk aversion, we must penalize a risk-averse agent for taking risky decisions, i.e. for investing significant amounts in the most volatile stocks. Moreover, we shall not forget about portfolio returns, R_t, when training our agent. To include both risk and return measures in our model, we use the quadratic utility function. Precisely, if the risk aversion coefficient is λ, the step reward is given by:

$$g_{t+1} = \log(1 + R_t - \lambda \bar{a}_t^\top \Sigma_t \bar{a}_t)$$

Example 3 - Utility optimization

A more general investment management problem, that is relevant not only for financial companies or investors but also for any corporation or individual, considers both - investment and consumption decisions. Consumption decisions refer to how much money to spend for one's needs or pleasures on a periodic basis. On one hand, by consuming a certain amount of money, an individual gains satisfaction (i.e., utility) where the level of "happiness" received varies from one person to another. On the other hand, investment decision may produce large returns and increase total wealth so that one can consume larger amounts in the future. Eventually, the goal of an agent is to maximize its total utility of consumption over a finite time period by making optimal investment and spending decisions. In the real world, each person/corporation has different external sources of constant or uncertain income and different spending needs, which makes the general problem of asset-allocation and consumption an extremely complex MDP. Consequently, we will first describe a simplified version of the problem and later on, introduce some realistic complexities.

We consider an individual that has a remaining lifetime of T time periods. This individual will not receive any income during that time, other than money extracted from the investment portfolio. The investment portfolio consists of N stocks and one riskless asset. Moreover, the agent will not have any fixed payment requirements (e.g., mortgage, essential food, insurance) such that all of its consumption is optional (e.g., vacation, leisure activities) and can be any real non-negative number. All of the capital extracted from the investment portfolio is immediately consumed and since there are no other sources of income, no capital is added to the investment portfolio at any point in time.

The agent starts with a dollar amount of W_0 and wants to maximize its expected aggregate utility of consumption over the whole time horizon T. At any time $t \leq T$, we note the amount spent on consumption by $c_t \geq 0$ and the portfolio's market value (or total wealth) by $W_t > 0$. The utility of consumption is given by a function $\mathcal{U} : \mathbb{R} \to \mathbb{R}$ such that by consuming c_t, an individual receives $\mathcal{U}(c_t)$ of satisfaction during period t. The utility function will typically include a parameter λ, standing for the risk aversion coefficient. We denote the portfolio allocation vector at time t as a_t where, as before, each constituent $a_t^{(j)}$ stands for the fraction of wealth allocated to asset $j = 0, 1, ..., N$. Note that both, consumption c_t and allocation a_t, are functions of time and wealth. At any time t, the agent must choose an optimal consumption amount and an optimal portfolio allocation. Thus, in the RL framework, we define the action at time t as a vector (c_t, a_t). Similarly to previous examples, the state will consist of current stock price information and the covariance

matrix, and will be defined by $s_t = [p_t \quad Tech_t \quad \Sigma_t]$. In addition, we include transaction costs defined as before by δ_j^+ and δ_j^- such that wealth evolution is:

$$W_{t+1} = (W_t - c_t)\,(a_t(1 + r_t))\,(1 - \psi(u_{t+1}))$$

In the above equation, $\psi(u_{t+1})$ stands for the total amount of transaction costs, when trading a percentage u_{t+1} of wealth at time $t + 1$. Finally, the step reward is defined as:

$$g_{t+1} = \begin{cases} \mathcal{U}(c_t) & \text{if } t < T - 1 \\ \mathcal{B}_T \cdot \mathcal{U}(W_T) & \text{if } t = T - 1 \end{cases}$$

where $\mathcal{B}_T$ is a very small number that is used here for technical reasons. In finance, this quantity is known as the bequest function.

3.2 Model Architectures

We consider the two cases of *Example 1* as well as the problem of *Example 2*. The first problem is variance minimization, further noted as ***MinVar*** example, the second problem is return maximization in presence of transaction costs, further noted as ***MaxRet*** example, and the third problem combines the two baseline settings by aiming to maximize risk-adjusted return (quadratic utility) of a risk-averse investor in presence of transaction costs, further noted as ***MeanVar*** example. In our experiments we started by approximating the optimal V-function and the optimal policy by two separate multilayer perceptrons (MLPs) that are trained through backpropagation. MLPs are the most basic, fully connected neural networks that are composed of linear layers with corresponding activation functions. Each MLP consists of D input neurons where D corresponds to the dimension of the state vector. For ***MinVar***, both networks have two linear layers with 24 hidden neurons and *tanh* activation functions. For ***MaxRet*** and ***MeanVar*** examples, network architecture is the same but the number of hidden neurons is increased to 64. The Actor MLP outputs a Gaussian distribution that approximates the optimal stochastic policy while the Critic MLP has one output neuron for the value function at a given state. All of the MLPs are trained with a learning rate of 0.0004 and entropy coefficient for the loss calculation of 0.005. Note that the training period ranges from January 1, 2009 to August 31, 2018, while the trading/testing period starts on September 1, 2018 and ends at January 1, 2020.

For all examples, we have defined a specific environment following the OpenAI gym's interface[5]. Recall that an environment has three key components namely, the state space $\mathcal{S}$, the action space $\mathcal{A}$ and the time t reward function $g(s_t, a_t, s_{t+1})$. Moreover, we must specify the valid format of action and state parameters for each specific environment. In other words, we must specify whether spaces are continuous or discrete, indicate lower and upper bounds for possible action and state values, and define the dimension of $\mathcal{A}$ and $\mathcal{S}$ spaces. For the state space, the dimension is determined by the size of the feature matrix $\mathcal{F}_t$ (risky stocks' information) and the size of other information arrays that are included in state vector s_t such as the covariance matrix of asset returns Σ_t. Similarly, if actions define portfolio target weights, the shape of the action space is equal to the number of assets considered. For the ***MinVar*** example, the portfolio is composed of $N = 4$ Dow Jones risky stocks and the state matrix includes monthly closing price vector and returns' covariance matrix. For the ***MaxRet*** example, the portfolio is composed of $N = 4$ Dow Jones stocks and one riskless

[5]See Appendix D.2

asset with a constant annual rate of interest $r_f = 1.00\%$. The ***MeanVar*** portfolio is also composed of one riskless and four risky assets and is held by a risk-averse investor which risk aversion is $\lambda = 1.5$. Furthermore, in addition to Dow Jones stocks' closing prices, the feature matrix also includes $I = 4$ technical indicators: the Moving Average Convergence/Divergence (MACD), the Relative Strength Index (RSI), the Directional Movement Index (DX), and the Simple Moving Average (SMA). For clarity, key parameters that were used to define each environment are specified in Table 1 below. After the training and testing phases, we have obtained the optimal actions

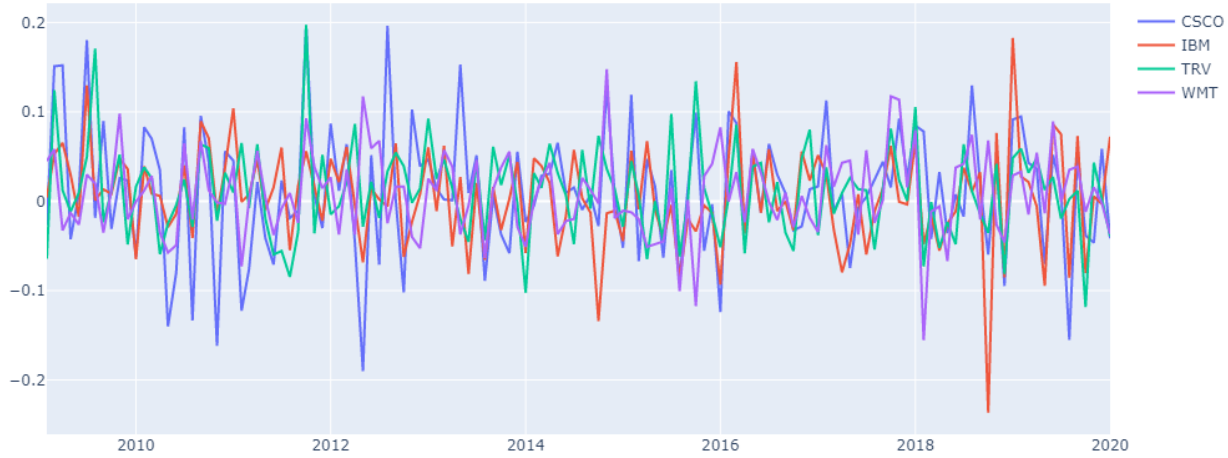
Table 1: Environment Details

Example	**MinVar**	**MaxRet**	**Risk Averse**
State	$[p_t \quad \Sigma_t]$	$[p_t \quad Tech_t \quad \Sigma_t]$	$[p_t \quad Tech_t \quad \Sigma_t]$
State Class	Continuous	Continuous	Continuous
State Dimension	N+ N×N	N+(I×N)+(N+1)×(N+1)	N+(I×N)+(N+1)×(N+1)
Action Space	$[0, 1]$	$[0, 1]$	$[0, 1]$
Action Class	Continuous	Continuous	Continuous
Actions Dimension	N	N+1	N+1
Initial capital ($)	$1M$	$1M$	$1M$
Fees per period (%)	None	$\delta^+ = 0.03$ $\delta^- = 0.02$	$\delta^+ = 0.03$ $\delta^- = 0.02$
Riskless Rate (%)*	None	1.0	1.0
Risk Aversion	None	None	1.5

*Annualized

and the resulting performance measures for each of the three models. In the plots that follow, we illustrate optimal portfolio allocations for four randomly selected Dow Jones stocks, namely: Cisco Systems Inc ($CSCO$), IBM Common stock (IBM), Travelers Companies Inc (TRV) and Walmart Inc (WMT). The monthly simple returns of these stocks over the combined training and trading periods are:

Figure 1: Stocks monthly returns



Over the whole period (training plus trading period), represented in figure 1, *CSCO* earned the highest mean monthly return, followed by *TRV* and *WMT* while the lowest return was earned by *IBM*. As for the risk, *CSCO* was the most volatile stock, followed by (*IBM*) and (*TRV*), while (*WMT*) was the less volatile. For the trading period, annualized mean returns and annualized volatility of returns are given in Table 2.

Table 2: Stocks Annual Returns and Volatility - Trading period

Ticker	**CSCO**	**IBM**	**TRV**	**WMT**
Annualized mean daily return	2.04%	-3.53%	5.84%	19.76%
Annualized volatility	25.70%	35.22%	17.50%	13.00%

The resulting optimal portfolio allocations (in percentage units) for the testing period are illustrated in figures 2, 3 and 4 below, for examples ***MinVar***, ***MaxRet*** and ***MeanVar***, respectively.

Figure 2: Optimal Allocation - MinVar

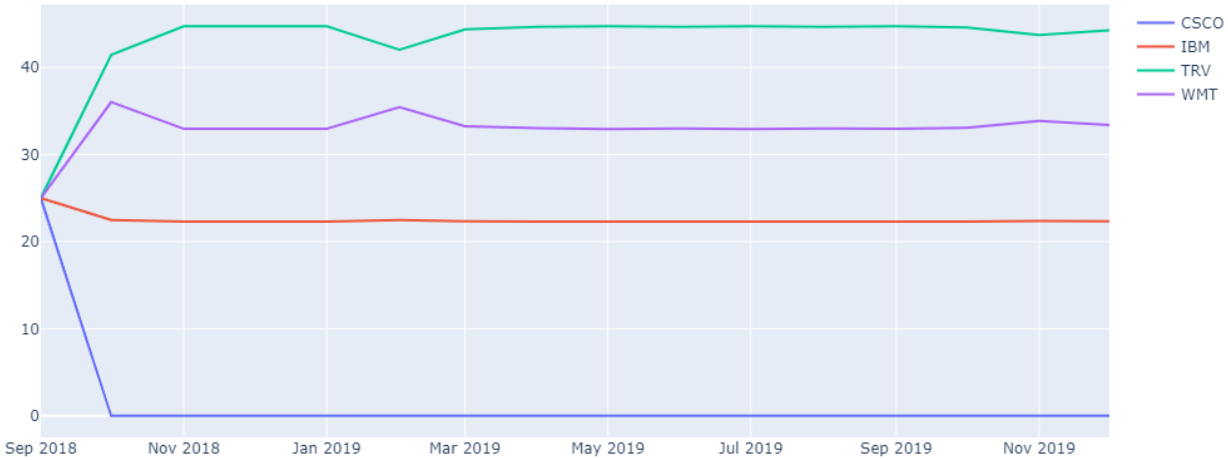


Figure 3: Optimal Allocation - MaxRet

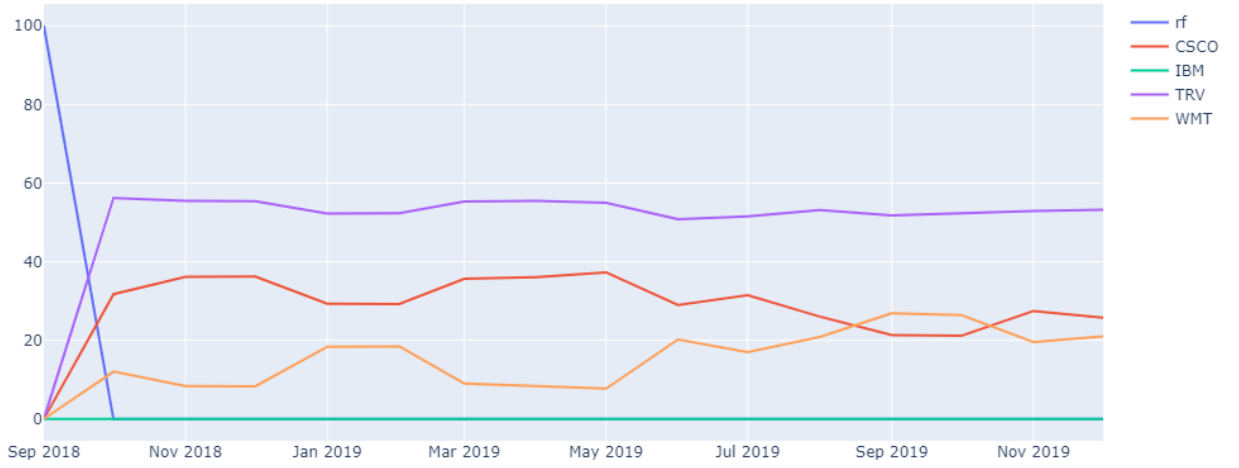
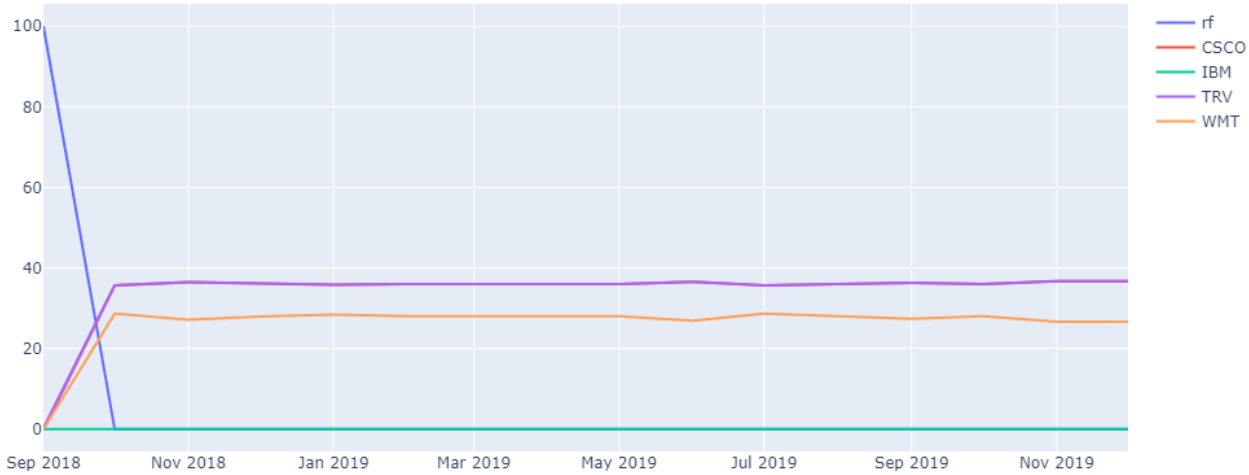


Figure 4: Optimal Allocation - MeanVar



Note that in figure 4, CSCO and TRV allocation curves are exactly the same. For clarity, we summarize the mean target weights for each of the baseline models in Table 2. Moreover, we compare the A2C MinVar model allocations to the analytical solution for the minimum variance portfolio.

Table 3: Target Allocation

Asset	**Risk-free**	**CSCO**	**IBM**	**TRV**	**WMT**
MinVar solution (%)	-	1.56	22.51	43.00	32.92
MinVar analytical solution (%)	-	5.40	15.96	45.08	33.56
MaxRet solution (%)	6.25	28.37	0.00	50.20	15.17
MeanVar solution (%)	6.25	33.86	0.00	33.86	26.04

By calculating the annualized return and volatility of each of the above A2C strategies as well as the terminal wealth at the end of the trading period, we have obtained the following results:

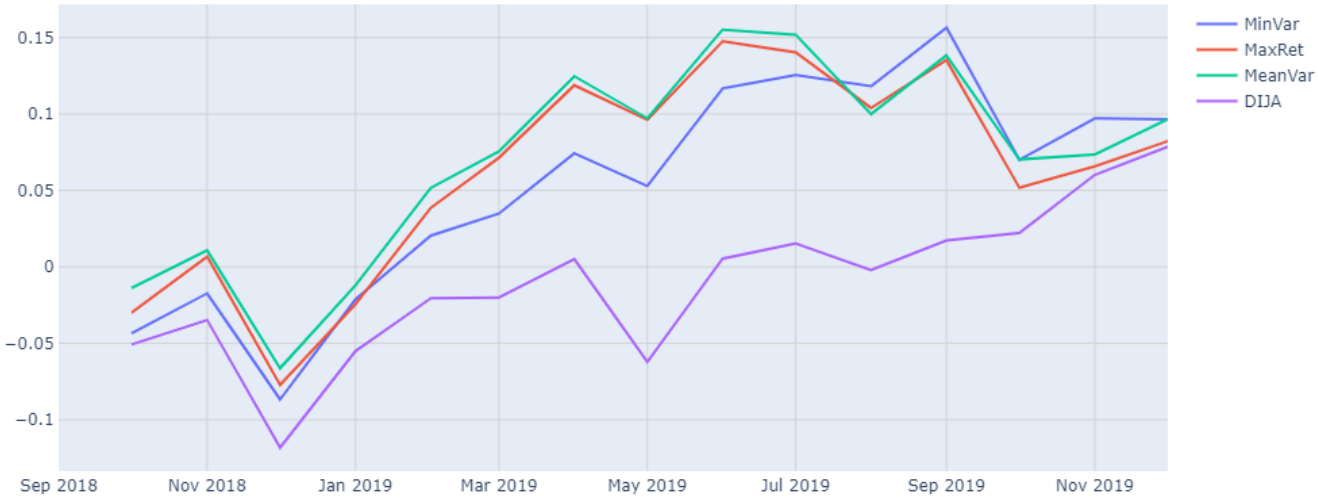
Table 4: Performance summary

	Mean return*	**Volatility***	**Terminal wealth**
MinVar portfolio	8.4846%	15.2376%	1.096 M
MaxRet portfolio	7.5299%	15.8635%	1.082 M
MeanVar portfolio	8.4785%	15.1463%	1.097 M

*Annualized

While the ***MinVar*** model obtains the highest annual return, the ***MeanVar*** agent obtained the best risk-adjusted return and the highest terminal wealth amount. Moreover, note that transaction costs were taken into account for the last 2 models which shows that the ***MeanVar*** model achieved similar annual return as the ***MinVar*** model even in presence of trading fees. Indeed, we see that in figure 4, there is little or no variation in allocation percentage over time meaning that the agent minimized the amount of transactions and thus, minimized transaction costs. This is opposed to figure 3 where curves are varying over the trading period which implies more transactions, higher fees and lower net return. Another observation is that while the ***MinVar*** agent invested almost 16% in IBM that has the highest volatility and negative mean return over the trading period, the ***MaxRet*** and ***MeanVar*** agents did not allocate any portion of its wealth to the IBM stock. In addition, baseline models allocated around $40 - 50\%$ of total wealth to TRV which is the second best stock in terms of risk-reward. This is different from the ***MeanVar*** model that has allocated approximately $1/3$ to both, TRV and $CSCO$. Finally, in Figure 5 below, we illustrate the cumulative returns obtained by each model as well as compare them to the cumulative returns of the Dow Jones index. Note that all A2C strategies outperformed the Dow Jones index on a cumulative return basis during the testing period.

Figure 5: Cumulative Returns



Appendix A Mathematical Proofs

A.1 Policy Gradient Theorem

To derive the Policy Gradient Theorem equation, we start by setting the gradient $\nabla_\theta \mathbb{E}[g_1] = 0$ to zero since the first action, a_0, does not depend on π_θ. Using the definition of $J(\pi_\theta)$ from section 1.4 and differentiating, leads to the below equation:

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \mathbb{E}_{\pi_\theta}\left[G(\tau)\right] = \sum_{t=1}^{T-1} \gamma^t \nabla_\theta \mathbb{E}_{\pi_\theta}\left[g_{t+1}\right]$$

$$= \sum_{t=1}^{T-1} \gamma^t \sum_{\tau_{0,t}} \mathbb{E}[g_{t+1}|s_t, a_t] \nabla_\theta P_{\pi_\theta}(\tau_{0,t})$$

Where

$$\begin{aligned}
\nabla_\theta P_{\pi_\theta}(\tau_{0,t}) &= \prod_{k=1}^{t} p(s_{k+1}|s_k, a_k) \nabla_\theta \left(\prod_{k=1}^{t} \pi_\theta(a_k|s_k) \right) \\
&= \prod_{k=1}^{t} p(s_{k+1}|s_k, a_k) \left(\sum_{s=1}^{t} \nabla_\theta \pi_\theta(a_s|s_s) \prod_{k=1, k\neq s}^{t} \pi_\theta(a_k|s_k) \right) \\
&= \prod_{k=1}^{t} p(s_{k+1}|s_k, a_k) \left(\sum_{s=1}^{t} \frac{\nabla_\theta \pi_\theta(a_s|s_s)}{\pi_\theta(a_s|s_s)} \prod_{k=1}^{t} \pi_\theta(a_k|s_k) \right) \qquad \text{by } \times \frac{\pi_\theta(a_s|s_s)}{\pi_\theta(a_s|s_s)} \\
&= \prod_{k=1}^{t} p(s_{k+1}|s_k, a_k) \pi_\theta(a_k|s_k) \sum_{s=1}^{t} \nabla_\theta \log \pi_\theta(a_s|s_s)
\end{aligned}$$

This implies that:

$$
\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \sum_{t=1}^{T-1} \gamma^t \left(\sum_{s_1,a_1} ... \sum_{s_t,a_t} \mathbb{E}_g[g_{t+1}|s_t,a_t] \prod_{k=1}^{t} p(s_{k+1}|s_k,a_k)\pi_\theta(a_k|s_k) \right) \sum_{s=1}^{t} \nabla_\theta \log \pi_\theta(a_s|s_s) \\
&= \sum_{t=1}^{T-1} \gamma^t \mathbb{E}_{\pi_\theta} \left[g_{t+1} \sum_{s=1}^{t} \nabla_\theta \log \pi_\theta(a_s|s_s) \right] \\
&= \mathbb{E}_{\pi_\theta} \left[\sum_{s=1}^{T-1} \sum_{t=s}^{T-1} \gamma^t g_{t+1} \nabla_\theta \log \pi_\theta(a_s|s_s) \right] \\
&= \mathbb{E}_{\pi_\theta} \left[\sum_{s=1}^{T-1} \nabla_\theta \log \pi_\theta(a_s|s_s) G(\tau_{s,T}) \right] \\
&= \sum_{s=1}^{T-1} \mathbb{E}_{\tau_{0,s}} \left[\nabla_\theta \log \pi_\theta(a_s|s_s) \mathbb{E}_{\tau_{s+1,T}}[G(\tau_{s,T})|s_s,a_s] \right] \\
&= \sum_{s=1}^{T-1} \mathbb{E}_{\tau_{0,s}} \left[\nabla_\theta \log \pi_\theta(a_s|s_s) Q^{\pi_\theta}(s_s,a_s) \right] \\
&= \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{s=1}^{T-1} \nabla_\theta \log \pi_\theta(a_s|s_s) Q^{\pi_\theta}(s_s,a_s) \right] \\
&\sim \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{s=1}^{T-1} \nabla_\theta \log \pi_\theta(a_s|s_s) Q^{\pi_\theta}(s_s,a_s)
\end{aligned}
$$

A.2 Multi-step TD error

The n-step TD error is, by definition, the sum of the one step TD errors discounted by γ. It follows that:

$$
\begin{aligned}
\hat{\delta}_t^{(n)} &= \sum_{s=0}^{n-1} \gamma^s \hat{\delta}_{t+s} \\
&= \sum_{s=0}^{n-1} \gamma^s \left(g_{t+s+1} + \gamma V_\nu(s_{t+s+1}) - V_\nu(s_{t+s}) \right) \\
&= \sum_{s=0}^{n-1} \gamma^s g_{t+s+1} + \gamma^n V_\nu(s_{t+n}) - V_\nu(s_t) \\
&= \sum_{k=t}^{t+n-1} \gamma^{k-t} g_{k+1} + \gamma^n V_\nu(s_{t+n}) - V_\nu(s_t)
\end{aligned}
$$

Appendix B Deep Neural Networks

A neural network is defined by a series of functional transformations, that is, non-linear compositions of functions of the form $f \circ g \circ h(x)$ where $x \in \mathbb{R}^d$ is a d-dimensional input to the network.

Such a structure is used to approximate functions of increasing complexity that will be able to map input to output.

To understand neural networks, we shall start by linear function approximations. A linear approximator consists of weights $\mathbf{W} \in \mathbb{R}^{n \times d}$ and a vector of bias $\mathbf{b} \in \mathbb{R}^n$. To simplify notation, we define $\theta = [\mathbf{W}, \mathbf{b}]$ to represent both, weights and bias. A linear approximator of a function $f(x)$ is obtained by an affine transformation of the input such that:

$$f(x) \approx \hat{f}(x; \theta) = \theta^\top x$$

The challenge is to find a value of parameters θ that will give the best estimator $\hat{f}$ of the target function f. In the simple, linear approximation case, components of θ are found in the same way as regression coefficients. In other words, we find θ^* that minimizes the mean squared error loss function between $f(x)$ and $\hat{f}(x; \theta)$.

Although linear models are useful, they have the downside of being restricted to the linear family of functions and thus, lack complexity to accurately understand complicated interactions between input variables. In order to increase the complexity of linear models, we introduce *at least* one non-linear function to the linear structure and as a result, obtain a DNN. The construction of a neural network starts by composing together different functions that will represent **layers** of the network. The simplest DNN is composed of an input layer, followed by a **hidden** layer, and finally, an output layer. Given an input vector $\boldsymbol{x}$ and noting by $h(\cdot)$ and $\sigma(\cdot)$ the hidden and output layers, respectively, the basic DNN has the following form:

$$f_\theta(\boldsymbol{x}) = \boldsymbol{\sigma}\left(\theta^{(2)} \boldsymbol{h}(\theta^{(1)} \boldsymbol{x})\right)$$

Input layer is the first layer receiving all the information contained in the input $\boldsymbol{x}$ and passing it to the successor hidden layers. Each neuron in the input layer is equal to the input feature/characteristic (e.g. asset price and/or asset type). The output layer gives the final output by transforming the output of the last hidden layer into the desired output type (e.g. probability, category, or real-valued number). Hidden layers are non-linear functions that are between the input and output layers. Hidden layers can be of any dimension (i.e. have any number of neurons).

Activation functions are used to add non-linearity and restrict the linear function output to a certain limit. Popular choices of activation functions are listed below.

- Rectefied Linear Unit (ReLU): $h(x) = \max\{0, x\}$
- Smooth ReLU: $h(x) = \log(1 + e^x)$
- Softmax (output activation): $\sigma(x_j) = \dfrac{\exp(x_j)}{\sum_i \exp(x_i)}$; limits the output to 0-1 range. Softmax is generally used when variables are categorical and we want to obtain their probability as output of a neural network.

To find optimal parameters θ of a neural network, the simplest approach is to, given an input x, minimize the error (loss) between the value predicted by the DNN, $f_\theta(\boldsymbol{x})$, and the true/target value associated with the given input, $f(x)$. Because we are dealing with expectations and the

distribution of data is typically unknown, empirical risk, $\hat{R}(\theta)$, is used as an approximation to the true loss, $L(x, f_\theta(x), f(x))$. The empirical risk is defined as:

$$\hat{R}(\theta) = \frac{1}{|\mathcal{D}|} \sum_{i=1}^{|\mathcal{D}|} L(x_i, f_\theta(x_i), f(x_i))$$

Where $\mathcal{D}$ is the dataset used during learning and $|\mathcal{D}|$ is the size of that dataset. Thus, the minimization of loss will be done through the minimization of the empirical risk, which in turn will be done by Gradient Descent. Finally, since the DNN, and thus the loss function, is a composition of functions, taking the gradient of the loss will involve the application of the Chain Rule and a concept called Backpropagation. For mathematical details on the computation of Gradient Descent and associated concepts, please refer to https://www.deeplearningbook.org/.

Appendix C Extensions of financial applications

C.1 Example 1 - Trades as actions

The portfolio is assumed to be self-financing in that all rebalancing of stock positions are financed from the cash account. At period t, the cash position is denoted b_t. We define by a vector h_t the dollar positions in each asset at the beginning of period t. Since short sales are not allowed, we must have $h_t^{(j)} \geq 0$ for all assets $j = 1, ..., N$. The state matrix includes the cash account balance, the stock positions, and the stock price data, i.e., it is defined as $s_t = [b_t \quad h_t \quad p_t \quad Tech_t]$.

Keeping the action space continuous, the vector $a_t = [a_t^{(1)} ... a_t^{(N)}]$ represents fractional trades to be made in period t. We denote by a_t^+ and a_t^- positive and negative trades with components $a_{t,j}^+ = \max(a_t^{(j)}, 0) > 0$ and $a_{t,j}^- = \min(a_t^{(j)}, 0) < 0$ standing for fractions of total wealth of asset j bought and sold, respectively. To simplify further calculations, we define a trade vector u_t to denote the dollar amount traded in period t. The trade vector u_t can be initially computed using $u_t = W_t \circ a_t$ where a_t is the raw output from the policy network, W_t is the total investor's wealth at the beginning of period t and $\circ$ denotes the element-wise product. Given the current state vector, the total wealth at t is calculated as the sum of the risky portfolio value and the cash account value, $W_t = h_t + b_t$. Note that because we cannot sell more shares than we own, we must have:

$$h_t + u_t \geq 0$$
$$\text{or}$$
$$u_t = \max(W_t \circ a_t, -h_t)$$

To compute the total transaction costs corresponding to the trading decision u_t, define by $\delta_j^+ \geq 0$ and $\delta_j^- \geq 0$ the proportional fees for buying and selling stock j, respectively. In addition, denote by $u_{t,j}^+ > 0$ and $u_{t,j}^- < 0$ the positive and negative trades for asset j. With this notation, the total value of transaction costs is obtained by:

$$\psi(u_t) = \delta^+ \cdot u_t^+ - \delta^- \cdot u_t^- = \sum_{j=1}^{N} \left(\delta_j^+ u_{t,j}^+ - \delta_j^- u_{t,j}^- \right)$$

Immediately after trades, the asset holdings are $h_t^+ = h_t + u_t$ and the next period holdings are computed with $h_{t+1} = (1 + r_t) \circ h_t^+$. Using the fact that the portfolio is self-financing, we compute the next step total wealth according to:

$$\begin{aligned} W_{t+1} &= P_{t+1} + b_{t+1} \\ &= \mathbb{I}^\top h_{t+1} + b_{t+1} \\ &= (1 + r_t)^\top (h_t + u_t) + (b_t - \mathbb{I}^\top u_t - \psi(u_t))(1 + r_f) \\ \text{subject to} \quad & b_t - \mathbb{I}^\top u_t - \psi(u_t) \geq 0 \end{aligned}$$

We initialize the cash balance to be equal the investor's initial capital, $b_0 = W_0$, implying that at time $t = 0$, the holdings are zero for all assets, $h_t^{(j)} = 0 \quad \forall j = 1, ..., N$. Because the objective is to maximize the portfolio value or return over the investment horizon, the terminal holdings are set to be zero, $h_T = 0$, signifying that all stock is converted to cash and $W_T = b_T$. Note that given the action space and transaction fees, it is possible that the agent runs out of cash, i.e., $b_t - \mathbb{I}^\top u_t - \psi(u_t) < 0$. If a cash shortage occurs, we can either break the training episode and penalize the agent, or simply not trade anything and continue the training until we have enough cash on hand. In both cases, we set the instantaneous reward equal to the daily profit made, $g_{t+1} = W_{t+1} - W_t$.

C.2 Example 2 - Trades as actions

We begin by the baseline example in the absence of transaction costs in order to define the baseline reward. Most of the definitions such as state, actions, and trade vectors, are kept from the example in Appendix *C.1* and we only modify the reward function. The next step total wealth obtained from selected trade u_t in excess of the risk-free growth is given by:

$$\begin{aligned} \Delta W_t &= W_{t+1} - (1 + r_f) W_t \\ &= (1 + r_t)^\top (h_t + u_t) + (b_t - \mathbb{I}^\top u_t)(1 + r_f) - (1 + r_f)(\mathbb{I}^\top h_t + b_t) \\ &= (r_t - r_f)^\top (h_t + u_t) \end{aligned}$$

The above result allows us to define the baseline component of the one-step reward:

$$g_{t+1}^{(0)} = \Delta W_t = (r_t - r_f)^\top (h_t + u_t)$$

Because in RL we maximize the expected value of total rewards, the expected baseline reward, given the state and action pair, is:

$$\mathbb{E}[g_{t+1}^{(0)} | s_t, u_t] = (\mathbb{E}[r_t] - r_f)^\top (h_t + u_t)$$

The expectation of stock returns, $\mathbb{E}[r_t]$, can be computed using past historical returns over a look-back period of one year (12 months or 252 days) from time t. Similarly, we can compute the covariance matrix of stock returns, Σ_t, that will be useful in the definition of risk penalty. This risk penalty, or the second component of the one-step reward, is given by:

$$g_{t+1}^{(risk)} = -\lambda \mathbb{VAR} \left(g_{t+1}^{(0)} | s_t, u_t \right) = -\lambda (h_t + u_t)^\top \Sigma_t (h_t + u_t)$$

Finally, the last component of the reward function is the transaction cost penalty, $g_{t+1}^{(tc)} = -\psi(u_t)$. Combining all the three reward/penalty components, we obtain that the expected instantaneous reward is:

$$\mathbb{E}[g_{t+1}|s_t, u_t] = \mathbb{E}[g_{t+1}^{(0)}|s_t, u_t] + g_{t+1}^{(risk)} + g_{t+1}^{(tc)}$$

Appendix D Coding and FinRL library

FinRL is a helpful and flexible tool for an easier implementation of RL methods in finance. It is an open source framework by *AI4 Finance-LLC* that covers the majority of DRL concepts defined in previous sections. Particularly, users can design their own RL environments, customize algorithms and DNN structures, and analyse algorithms' performance. FinRL makes use of *Yahoo Finance* to download all the necessary stock data and of *Stable Baselines3* to train and test algorithms such as A2C and PPO. In this section, we provide a quick guide on how to use these libraries while a more detailed description of FinRL and Stable Baselines3 can be found at https://finrl.readthedocs.io/en/latest/index.html and https://stable-baselines3.readthedocs.io/en/master/index.html.

D.1 Data

As has been mentioned, the data is downloaded through `yfinance` Python package. To do so, we must specify the start and end dates, and tickers (list of stocks' symbols) to then pass it to the `download_data` function of the `YahooFinanceProcessor` class. If we want to access tickers of stocks that are constituents of specific indices such as $S\&P500$ or DJI, we can use the `pytickersymbols` python package. After the price data is downloaded, we must clean the data and if required, add technical indicators or other stock features. If daily price data was retrieved, the cleaning process as well as the stock features addition can be done with the incorporated `FeatureEngineer` class. Before proceeding to the DRL step, the pre-processed data must be divided into the training and testing sets. The splitting is done by defining two mutually exclusive time periods (with different starting and ending dates), one for the training and one for the testing. Note that typically, the training data makes more than 70 percent of the whole stock dataset. In our examples, the training period is from January 1, 2009 to June 30, 2020 while the testing period is from July 1, 2020 to September 2, 2021. Note that because there is no trading during weekends and holidays, one year has approximately 252 entries of daily stock price information instead of 365. Given these points, we can proceed to the environment definition.

D.2 Environment

Modelling the environment of a specific problem is a key task in Reinforcement Learning. An environment contains all the necessary functionality for running and training the agent. In other words, it allows the agent to explore the processed data and translate the retrieved information into a MDP problem. FinRL library requires that we define the environment by strictly following the OpenAI gym's interface (please see https://www.gymlibrary.ml/ for more details). It must include, at minimum, the following methods:

- `init`: defines the state and the action spaces as well as constant variables which will be used in the environment class functions (e.g., data, initial capital, transaction costs). The state

space must be represented by the `observation_space` attribute which defines the structure and the acceptable range of values that a state of the environment can take. Similarly, the `action_space` attribute describes the legitimate actions that can be applied to the environment. There are different categories of spaces available, but the two classes that will be used for our examples are `Box` and `Dict` spaces. States or actions that are represented by a `Box` space take on real values, and can vary in shapes. The shape of a space corresponds to its dimension, that is, if the action space $\mathcal{A}$ is continuous and represents weights of a long-only portfolio composed of N stocks, the shape of $\mathcal{A}$ will be `(N,)`. Comparatively, states or actions that are represented by a `Dict` space are also real-valued, but can represent compound spaces, i.e., spaces that are composed of different sub-spaces. As an example, consider a portfolio of N stocks and one risk-free asset. Assume that the state s_t is composed of past portfolio allocation a_{t-1} and stock features matrix $\mathcal{F}_t$ with I different technical indicators for N stocks. The `observation_space` can be defined as a dictionary of two `Box` spaces: a space of shape `(N+1,)` to represent past actions and a space of shape `(I,N)` to represent stock feature matrix. Note that in the case of a `Dict` space, the policy will be a *Multi-input* policy since its input will be a state composed of multiple spaces.

- `reset`: resets the environment to the initial state and returns that initial state.
- `step`: given an action as an input, it applies that action to the environment and makes the agent transition to a new state. The function must return the new state, the instantaneous numerical reward and the boolean indicator of whether the agent has reached the terminal state. The output can also include a dictionary containing additional information that may be useful in debugging.
- `save_asset_memory`: keeps in memory the portfolio information which could be accessed during the testing stage. Such information is stored in a data frame and may include portfolio returns, value of wealth, and rewards over the complete investment horizon.
- `save_action_memory`: similar to `save_asset_memory` but records optimal actions that were made at every time step during the testing phase.

D.3 DRL

Having a well defined environment, we can proceed to the reinforcement learning process. The agent layer in FinRL, which is defined by the `DRLAgent(environment)` method, is responsible for interacting with the defined environment. When the environment with its key arguments is passed to the DRL agent, we must select a specific algorithm (e.g., A2C) and choose corresponding hyperparameters (e.g., number of episodes, learning rate, network architectures). Algorithm specific hyperparameters are passed in a dictionary which specifies some or all of the following arguments:

- `model_name`: to specify the Stable Baselines3 algorithm to implement; for the Actor-Critic algorithm, the model name is set to "a2c".
- `policy`: to specify the policy model to use; readily availbale specifications include "MlpPolicy", "CnnPolicy", and "CnnLstmPolicy" however, it is possible to define a customized policy network which must follow the Stable Baselines3 interface; the default value is "MlpPolicy".
- `gamma`: to specify the discount factor for instantaneous rewards; the default value is 0.99.

- `n_steps`: to specify the number of steps to run per update; the default value is 5.
- `vf_coef`: to specify value function coefficient when calculating the total loss; the default value is 0.25.
- `ent_coef`: to specify entropy function coefficient when calculating the loss; the default value is 0.01.
- `learning_rate`: to specify the learning rate used for both, value and policy functions; the default value is 0.0007.
- `max_grad_norm`: to specify the maximum bound for the gradient norm which is used in gradient clipping in order to limit extremely large parameters updates; the default value is 0.5.
- `policy_kwargs`: arguments to be passed for the network creation including a list `net_arch` which specifies the value and policy network architectures, and `act_fun`, an activation function to use in the neural network; the default values are `net_arch=[dict(pi=[64, 64], vf=[64, 64])]` and `act_fun=torch.nn.tanh`. For a shared, three-layered network where each layer has 72 hidden units, we must set `net_arch=[72, 72, 72]`; for a fully separate network where the policy network has two layers of size 72 and the value network has three layers of size 64, we must set `net_arch=[dict(pi=[72, 72], vf=[64, 64, 64])])`.

The above arguments are passed to the `DRLAgent.get_model()` method to define a specific model which in turn, is passed to the `DRLAgent.train_model(model, tb_log_name, total_timesteps)` function in order to begin training. The argument `tb_log_name` specifies how to print information regarding the RL agent during the training phase and makes it possible to observe the evolution of loss functions, rewards, completed updates, etc. The `total_timesteps` argument specifies the total number of runs such that `total_timesteps = n_steps` $\times$ `number of updates`. Note that the obtained trained model, defined by `trained_a2c`, can be saved in some directory of choice which allows to access the model later on an apply it to different datasets. To save the model, simply choose a file path and run the `trained_a2c.save(path)` command. When you would want to access the trained model again, without going through the training process, the `model_sb3_type.load(path)` command should be executed, where `model_sb3_type` refers to one of the Stable Baselines3 model types (A2C, PPO, DDPG, SAC, DQN, etc).

Finally, using the test dataset, we run the `trained_a2c.predict()` command to sample optimal actions at each encountered state and the `environment.step()` command to make a transition to the next state. These commands are executed until the end of the testing time-horizon. When the agent reaches the terminal state, we could obtain the portfolio information and the optimal actions over the whole testing period by calling the `save_asset_memory` and `save_action_memory` methods defined previously.

D.4 Comparison and Plots

To plot the results, the package `plotly` is used. Since after testing we can access the optimal actions and specific portfolio information, we start by plotting optimal actions for different models using the `add_trace` command of the `plotly.graph_objs.Figure()`. In all plots, the x-axis

represents the testing dates and the y-axis represents numerical values of optimal actions or of portfolio characteristics.

In addition, we could compare the performance of trained DRL strategies to benchmarks (such as Dow Jones and S&P indices) or to classic portfolio strategies (such as minimum variance and maximum sharpe portfolios). An easy and fast way to obtain the performance of the latter strategies for the same data that was used in the DRL algorithm, is to use the package `PyPortfolioOpt`. The package includes a class called `EfficientFrontier` that contains different optimization techniques, each corresponding to diverse objective functions. Below, we list the key inputs to the `EfficientFrontier` class.

- `expected_returns`: an array, series, or list of annualized expected returns for each asset; can be `None` if the strategy only optimizes for volatility measure. Since we deal with historical data, at each time-step t, we considered the previous 252 daily returns and used them to compute the annualized mean returns for each asset.
- `cov_matrix`: an array or data frame that contains covariance of returns for each asset; must be provided and must be positive semidefinite. At each time-step and for each asset, we computed the covariance of the previous 252 daily returns.
- `weight_bounds`: a tuple or list of allowed weight allocation bounds for each or all assets; for a long-only portfolio (default) the bounds are the same for all assets and the value is set to `(0,1)`; for a portfolio where short positions are allowed, the value is set to `(-1,1)`.

As mentioned earlier, multiple methods are available for different investment objectives. In our comparison we consider two objectives, namely, minimise the portfolio volatility and maximize the Sharpe ratio. For the former objective, we use the `min_volatility()` method; for the latter objective we use the `max_sharpe(risk_free_rate)` method where we must specify the riskless rate of return over a period that corresponds to the frequency of the `expected_returns` input. Both methods return an ordered dictionary of selected asset weights for the corresponding optimal portfolio. It follows that at each testing date, we compute the optimal weights, multiply them by the current portfolio value and then, by observing the next step closing prices, we accumulate the current wealth with one period asset returns. At the end of the testing horizon, we will obtain the evolution of wealth corresponding to each of the efficient frontier strategies. Using the plotting tools mentioned at the beginning of this subsection, we have all the necessary information to compare the performance of the selected strategies.

4 References

[1] Danthine, J-P., Donaldson J.B. (2015). *Chapter 3 of Intermediate Financial Theory, Third Edition.* Academic Press.

[2] Bertsekas, D. P. (2005). *Dynamic programming and Optimal Control.* Athena Scientific.

[3] Rao, A., Jelvis, T. (2005). *Foundations of Reinforcement Learning with Applications in Finance.* Athena Scientific.

[4] Sutton, Richard S., Andrew G. Barto. (2018). *Reinforcement Learning: An Introduction.* Second. The MIT Press.

[5] Mnih, V., Badia, A.P., Mirza, M. and al. (2016). *Asynchronous Methods for Deep Reinforcement Learning.* https://arxiv.org/pdf/1602.01783.pdf

[6] Bergdahl, J. (2017). *Asynchronous Advantage Actor Critic with Adam Optimization and a Layer Normalized Recurrent Network.* KTH Royal Institute of technology.https://www.diva-portal.org/smash/get/diva2:1169944/FULLTEXT01.pdf

[7] Dellinger, J. (2019). *Weight Initialization in Neural Networks: A Journey From the Basics to Kaiming.* https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-fro