The Cooper Union for the Advancement of Science and Art
Albert Nerken School of Engineering

A Deep Reinforcement Learning Approach to the Portfolio Management Problem

Sahil S. Patel

A thesis submitted in partial fulfillment
of the requirements for the degree
of Master of Engineering

April 11, 2018

Professor Sam Keene, Advisor

The Cooper Union for the Advancement of Science and Art
Albert Nerken School of Engineering

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

Richard Stock, Dean of Engineering Date

Prof. Sam Keene, Thesis Advisor Date

Acknowledgements

Sam Keene for being a constant help throughout my Cooper career, especially over this past year as my thesis advisor.

Chris Curro for being an extraordinary guide, resource, and friend.

Yash Sharma for helping me get through many of my blocks.

My family for always supporting me.

Abstract

Reinforcement leaning attempts to train an *agent* to interact with their *environment* so as to maximize its expected future *reward*. This framework has successfully provided solutions to a variety of difficult problems. Recent advances in deep learning, a form of supervised learning with automatic feature extraction, have been a significant factor in modern reinforcement learning successes. We use the combination of deep learning and reinforcement learning, deep reinforcement learning, to address the portfolio management problem, in which an agent attempts to maximize its cumulative wealth spread over a set of assets. We apply Deep Deterministic Policy Gradient, a continuous control reinforcement learning algorithm, and introduce modifications based on auxiliary learning tasks and $n - step$ rollouts. Further, we demonstrate its success on the learning task as compared to several standard benchmark online portfolio management algorithms.

Contents

List of Figures

1 Background

1.1 Machine Learning Fundamentals

We first describe machine learning fundamentals using the notation developed by Goodfellow et al. (2016) [3] and Bishop (2006) [1]. Machine learning encompasses algorithms that aim to learn to perform tasks T from experience E. T can range through many different tasks, such as:

- Classification - the task of determining which category an item x is. This function can be represented as $f : \mathbb{R}^n \to \{1, ..., k\}$, where $x \in \mathbb{R}^n$, and k is the total number of categories the model is aware of.

- Regression - the task of determining a numerical value given an input x. This function can be represented as $f : \mathbb{R}^n \to \mathbb{R}$, where $x \in \mathbb{R}^n$.

Many other tasks exist, however classification and regression are two of the most common types of tasks that machine learning systems are often built to perform.

We measure the performance of the machine learning system by a performance metric that is specific to the task that the system performs. For classification, one could use the accuracy of the model, while for regression one could use the mean-squared error between the model's predictions and the labels in the dataset.

When training a machine learning model, we are mainly concerned about its *generalization* ability, or its ability to perform on previously unseen inputs. We can approximate this error by splitting our dataset into two portions: a *training* set and a *testing* set. We limit our model to only utilizing on the training set to perform predictions and we estimate its generalization ability using the test set. Many models, however, have *hyperparameters* that are not learned through training; instead, these are set at the beginning before training. We can introduce a third split, the *validation* set to choose these parameters.

We expect our model to generalize to the test set because of the *independently and identically distributed (i.i.d.) assumptions*, where we assume that the test set and training set were produced by the same *data generating process*. By reducing the error of our model on the training set, we expect our error on the testing set to be reduced as well. Two problems can occur however. The model can either *underfit* when it does not have enough *capacity* to fit the training set, and thus its training error will be large. On the other hand, the model can *overfit* when it has too much capacity, and thus memorizes features of the training set that may not necessarily apply on the test set. In this case, the gap between the training error and test error will be large [3]. This is visualized in Figure 1. What we have described thus far

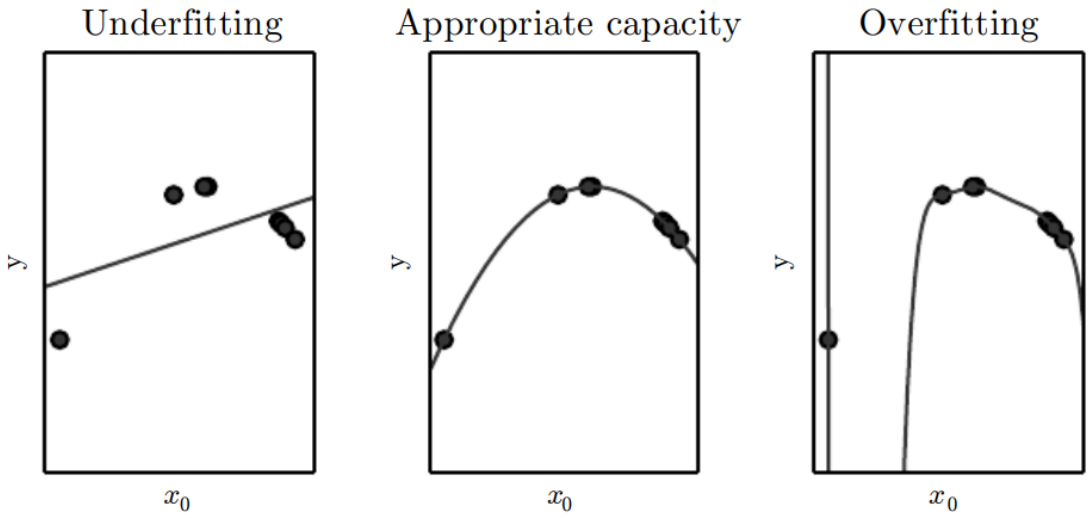

Figure 1: Capacity of a model and its impact on generalization [3]

is the concept of *supervised learning*. Two other subcategories of machine learning exist, *unsupervised learning* and *reinforcement learning*. Let us provide definitions of all three:

- Supervised learning - learn to predict an outcome using labeled data
- Unsupervised learning - learn the underlying structure of data

- Reinforcement learning - learn how to behave given a reward signal and state information

1.2 Reinforcement Learning (RL)

1.2.1 The Framework

In stark contrast to supervised learning, the only signal that an agent trained through RL receives is the *reward signal* that indicates a objective change of the satisfaction level of the agent's current *state* [39]. Furthermore, unlike supervised learning, this reward signal can be delayed and samples of the reward signal are inherently sequential and nonstationary. Although reinforcement learning does not fall within the supervised learning umbrella, it cannot be categorized as a type of unsupervised learning. The major task of unsupervised learning is to find inherent structures within a dataset – and while this can be useful for maximizing an agent's cumulative reward – it is not the ultimate task of a reinforcement learning agent. These complications require us to build a framework separate from that of widely developed supervised and unsupervised learning methods[32]. The framework should allow the agent to learn solely from interaction, and therefore must be broader than standard supervised learning techniques.

Let us first consider the three essential components of the framework: the reward signal, agent, and environment. The reward signal, $R_t \in \mathbb{R}$, indicates how well the agent is performing at time step t. The goal of the agent, at time step t, is to perform actions optimally such that the cumulative reward (the return) it receives from t onwards, G_t, is maximal. *Reinforcement* learning attempts to guide an agent to act optimally within its environment, and has no fixed dataset, since its dataset consists of the experience the agent has gained from interacting with its environment.

$$G_t = \sum_{i=t}^{\infty} R_{i+1} \tag{1}$$

Therefore, we must assume that solutions to problems we wish to solve can be translated to maximizing an agent's cumulative reward.

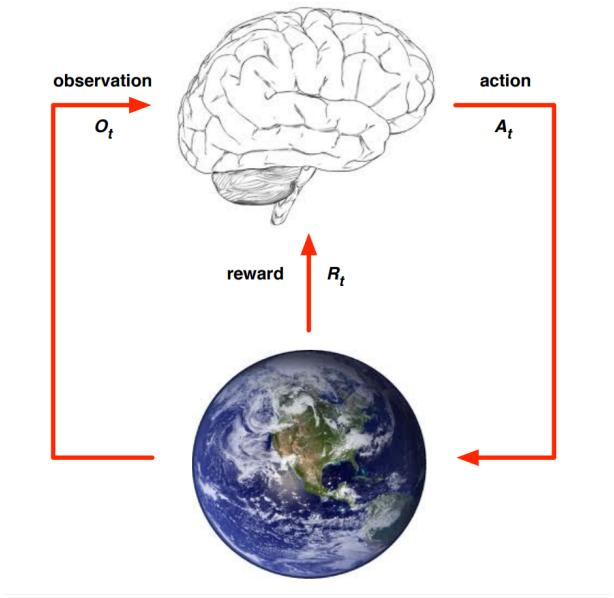

Figure 2: Interaction between agent and the environment [32]

Figure 2 characterizes the interactions of the agent (depicted by the brain), the environment (portrayed by the Earth), and the reward signal within the reinforcement learning framework. The agent performs actions, A_t, that alter its state in its environment. It is then able to make observations, O_t, that provide information regarding its state. Lastly, the agent receives a reward signal, R_t, from the environment that indicates how well the agent is performing.

1.2.2 Markov Decision Processes

At all time steps, the agent is aware of the history of information it has received:

$$H_t = \{O_1, R_1, A_1, ..., A_{t-1}, O_t, R_t\}$$

Based on this history, the agent is able to choose the next action it believes will maximize its cumulative reward. It is impractical to expect an agent to utilize the

entire history to base its next action upon, as the memory costs associated with such an agent would become enormous. Therefore, we introduce the notion of *state*, the agent now utilizes the state instead of the history to influence its decision. The state at a certain time step should thus capture all relevant information of the history:

$$S_t^a = f(H_t)$$

for example,

$$S_t^a = O_t$$

It is important to note that the state can be *any* function of the history (not just simple ones as shown above), as long as it is able to distill relevant information [32]. Later we will demonstrate the usage of neural networks in extracting state information from a history sequence.

We now make the simplifying assumption that the distribution of states has the *Markov* property:

$$\mathbb{P}[S_{t+1}^a | S_t^a] = \mathbb{P}[S_{t+1}^a | S_1^a, ..., S_t^a]$$

The environment, on the other hand, maintains its own state as well, S_t^e. The environment's state, unlike the agent's state is *defined* to be Markov, rather than *assumed*. To illustrate the differences, an environment state could be a video game's internal state that determines the future dynamics of the player's experience, while an agent's state would be what the player perceives their current placement in a game to be.

If the agent has direct access to the environment state, it is in a *fully observable* environment:

$$S_t^a = S_t^e$$

If the agent does not have direct access to the environment state, such as a blackjack

agent that can not view the dealer's cards, it is in a *partially observable* environment:

$$S_t^a \neq S_t^e$$

Given our current notion of a Markov state that contains all relevant information of the history, the agent is now able to base actions upon the state it believes itself to be in. We assume henceforth that $S_t = S_t^a$. The *policy* function maps agent states to actions:

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$$

When an agent performs an action in a given state, there are many states the agent could end up in due to factors present in the environment. The transition dynamics, $\mathcal{P}$, describe the distribution of future states the agent could end up in given its current state and action choice:

$$\mathcal{P}_{ss'}^a = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a]$$

Similarly, there are many rewards the agent could receive upon acting a certain way in a given state. The reward function, $\mathcal{R}$, governs this distribution:

$$\mathcal{R}_s^a = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$$

We now have the facilities to describe reinforcement learning problems through *Markov Decision Processes* (MDPs), which are defined by tuples of the form $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$ [33]:

- $\mathcal{S}$ is the set of all states the agent could be in
- $\mathcal{A}$ is the set of all actions the agent can perform
- $\mathcal{P}$ is the transition dynamics of the environment, $\{P_{ss'}^a \quad \forall a \in \mathcal{A}\}$

- $\mathcal{R}$ is the reward function of the environment, $\{R_s^a \quad \forall a \in \mathcal{A}\}$
- γ is the *discount factor* $\in [0, 1]$ that governs how much the agent weighs future rewards received. We modify Eq. 1 for the return at time step t as follows:

$$G_t = \sum_{i=t}^{\infty} \gamma^{i-t} R_{i+1} \tag{2}$$

Given a fully defined MDP, our problem statement reduces to determining the optimal policy by which the agent should perform actions. We define two quantities, the *state-value function*, $v_\pi(s)$, and the *action-value function*, $q_\pi(s, a)$ to aid us in determining the optimal policy, π.

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] \tag{3}$$

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \tag{4}$$

$v_\pi(s)$ is the expected return of following policy π from the starting state s. $q_\pi(s, a)$ is the expected return of following policy π after having taken action a from the starting state s [33].

1.2.3 The Bellman Equations

The state and action value functions can be decomposed in terms of themselves:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \tag{5}$$

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \tag{6}$$

Eqs. 5 and 6 form the *Bellman Equations* for state and action-value functions, which can also be decomposed in terms of each other [35]:

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) q_\pi(s, a) \tag{7}$$

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v_\pi(s') \tag{8}$$

We can combine these equations to yield [35]:

$$v_\pi(s) = \sum_{a \in A} \pi(a|s)[\mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v_\pi(s')] \tag{9}$$

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a [\sum_{a' \in A} \pi(a|s') q_\pi(s',a')] \tag{10}$$

Eqs. 9 and 10 form the *Bellman Expectation Equations* and allow us to evaluate the state and action value functions of a given policy π for a given MDP [39]. What we want, however, is the policy π that maximizes the state and action value functions:

$$v_*(s) = \max_\pi v_\pi(s)$$

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

Once we have these optimal value functions, our agent can select actions optimally by simply selecting the action that corresponds to the maximum action-value:

$$\pi_*(a|s) = \begin{cases} 1 & a = \arg\max_{a \in A} q_*(s,a) \\ 0 & else \end{cases}$$

How do we go about finding $q_*(s,a)$ (and thus, $v_*(s,a)$)? We can use Eqs. 7 and 8 to describe $v_*(s)$ and $q_*(s,a)$ in terms of each other:

$$v_*(s) = \sum_{a \in A} \pi_*(a|s) q_*(s,a)$$

$$v_*(s) = \max_{a \in A} q_*(s,a) \tag{11}$$

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v_*(s') \tag{12}$$

Once again, we can combine Eqs. 11 and 12 together to form the *Bellman Optimality Equations*:

$$v_*(s) = \max_{a \in A} \mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v_*(s') \tag{13}$$

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a \max_{a' \in A} q_*(s',a') \tag{14}$$

Unlike the Bellman Expectation Eqs. 9 and 10, the Bellman Optimality Equations are nonlinear (due to the max operation present) and thus cannot be solved by a simple matrix inverse. We must therefore use iterative methods to find the optimal value functions.

1.2.4 Dynamic Programming

We now consider methods that determine optimal policies *given* a complete representation of the environment through an MDP. First, we must be able to determine v_π for any policy π – this step is called *policy evaluation*. We can then use v_π to evolve our current policy to π', such that $v_{\pi'}(s) \geq v_\pi(s), \forall s \in \mathcal{S}$ – this step is termed *policy improvement*. We can evaluate π' and improve upon it – the interleaving of policy evaluation and policy improvement is termed *policy iteration*. Using policy iteration to reach the optimal policy π_* is *solving* the MDP through *dynamic programming* [39].

Let us begin with policy evaluation by considering the Bellman Expectation Eq. 9 again. If we write:

$$\mathcal{P}_{s,s'}^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}_{ss'}^a$$

$$\mathcal{R}_s^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}_s^a$$

essentially averaging the transition dynamics and reward function over all actions, we can simplify the Bellman Expectation Equation as follows [35]:

$$v_\pi = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v_\pi$$

We can then solve for v_π, or *evaluate* policy π, with an $O(N^3)$ policy evaluation solution for N states via a matrix inverse:

$$v_\pi = (I - \gamma \mathcal{P}^\pi)^{-1} \mathcal{R}^\pi$$

This solution, unfortunately, cannot be used for MDPs with large state spaces because of the large runtime, so we turn to an iterative policy evaluation algorithm instead. If we consider an initial approximation of the state value function for all states, v_0, and consider a sequence of repeated approximations by applying the Bellman equation (Eq. 5), $\{v_1, v_2, ..., v_n\}$, v_n will converge to v_π, as shown in [39]. Given this convergence guarantee, we can use the following to update our approximation of the value function:

$$v_{n+1}(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_n(S_{t+1}) | S_t = s] \tag{15}$$

Now we address policy improvement by considering the Bellman Eq. 6. We want to form a policy π' such that $v_{\pi'}(s) \geq v_\pi(s), \forall s \in \mathcal{S}$. Note that since

$$v_\pi(s) = \mathbb{E}_\pi[q_\pi(S, A) | S = s]$$

if we choose action $a' = \arg\max_{a \in \mathcal{A}} q_\pi(s, a)$, then $q_\pi(s, a') \geq v_\pi(s)$. If we perform this greedy maximization on every state, we end up with a new policy π' that fulfills our requirements for policy improvement. Given both policy evaluation and improvement techniques, we can simply interleave these operations to form the *policy iteration* algorithm. A special case of the policy iteration algorithm exists where we only perform one iteration of policy evaluation (rather than waiting for our approximations to converge), this case is termed *value iteration*.

Figure 3 depicts how policy iteration brings us from a sub-optimal policy and an approximate value function to the optimal policy and value function.

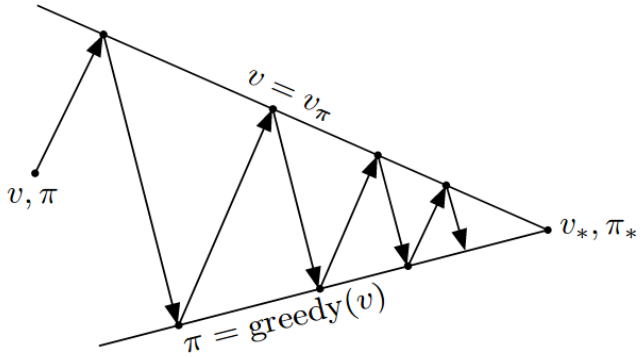

Figure 3: Policy iteration [39]

1.2.5 Learning from Experience

The policy iteration method discussed in the previous section only allowed us to find an optimal policy *given* a fully defined MDP. This constraint, however, is unlikely to be satisfied in real environments, which are often extremely complex, leaving us unable to determine the transition dynamics and the reward function of the environment. Therefore, an agent must be able to learn *solely* from experience if it is to be of practical use. Similarly to dynamic programming, we will approach this problem by considering policy evaluation, improvement, and iteration algorithms that are model-free.

1.2.5.1 Model-Free Prediction

We first consider policy evaluation algorithms, which, in the context of model-free algorithms, are termed *model-free prediction* algorithms. Recalling Eqs. 3 and 4 which define the state and action-value functions, respectively:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

it is clear that a simple way to estimate both value functions is through a *Monte-Carlo* approach. We can sample *episodes* from the environment using policy π to guide our agent:

$$\mathcal{E} = \{S_1, A_1, R_2, ..., S_k\} \sim \pi \tag{16}$$

where S_1 is an initial state and S_k is a terminal state. We can determine the return, G_t, for each state $s \in \mathcal{E}$ or for each tuple $(s, a) \in \mathcal{E}$. By running multiple episodes, we can average the returns experienced for each state or state action pair, directly approximating the value functions. Pseudocode that implements this approximation of the state value function is shown in Algorithm 1. Since Monte-Carlo prediction

Algorithm 1 First-visit MC prediction of v_π [39]

procedure MCVPREDICTION(π, N)
    $V(s) \leftarrow 0 \quad \forall s \in \mathcal{S}$
    $Returns(s) \leftarrow$ an empty list $\forall s \in \mathcal{S}$
    $n \leftarrow 0$
    repeat
        Generate an episode $\mathcal{E}$ using π
        for $s \in \mathcal{E}$ **do**
            $G \leftarrow$ return following the first occurrence of s
            Append G to $Returns(s)$
            $V(s) \leftarrow$ average($Returns(s)$)
        end for
        $n \leftarrow n + 1$
    until $n = N$
    Output $V(s)$
end procedure

directly estimates the value functions using their definitions, it is guaranteed to converge correctly. On the flipside, it requires episodes to be run to *completion* in order to calculate the returns G. Furthermore, MC prediction is a high variance method, and thus takes many iterations to converge. We can trade off some of these disadvantages using a method called *Temporal-Difference* (TD) learning, which performs biased updates to our estimate of the value function.

MC prediction essentially updates our state value function estimate, $V(S_t)$, to-

wards the actual return, G_t, which is an unbiased estimate of $v_\pi(S_t)$:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$$

where for Algorithm 1, $\alpha = \frac{1}{N(S_t)}$, where $N(S_t)$ is the number of times state S_t was encountered. TD learning, instead, updates our state value function estimate towards the *TD target*, $R_{t+1} + \gamma V(S_{t+1})$:

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

Since TD learning uses our value function estimate to formulate the TD target, it is called a *bootstrapping* method [36]. The TD target in this case is bootstrapped after one signal of reward, so this method of learning is called *one-step* TD learning, or TD(0). Since TD learning only requires the reward R_{t+1} and not the return G_t, it can be applied in an online fashion without requiring episodes to be completed.
Furthermore, considering the variance of the TD target in comparison to the MC target, it is clear that:

$$\text{VAR}[R_{t+1} + \gamma V(S_{t+1})] \leq \text{VAR}[G_t] = \text{VAR}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...]$$

and therefore, TD learning trades off some of the variance of MC prediction for biased updates, which can lead to faster learning in stochastic environments. Other versions of TD earning exist, such as TD(λ), where the λ parameter controls the level of bootstrapping we wish to use [36]. In TD(λ), the TD target is now:

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

where $G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$ is the *n-step return*. A λ value closer to zero reduces our variance, while a value closer to one reduces our bias. Note

that while model-free prediction of the state value function, v_π was discussed, analogous results apply to the prediction of the action value function, $q_\pi(s, a)$. Specifically, for MC prediction, we use the update:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t))$$

where $Q(S_t, A_t)$ is our action value function estimate and $\alpha = \frac{1}{N(S_t, A_t)}$, where $N(S_t, A_t)$ is the number of times the tuple (S_t, A_t) was encountered. For TD learning, specifically TD(0), we use:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

1.2.5.2 Model-Free Control

Now that we have methods to evaluate an agent's policy, we can continue along the policy iteration framework developed for dynamic programming. Here we consider both policy improvement and policy iteration, which fall under the umbrella of *model-free control* algorithms when we learn entirely from experience.

In the dynamic programming case, we were able to simply construct an improved policy by greedily maximizing over the action-value function, $\pi'(s) = \arg\max_a q_\pi(s, a)$. In practical scenarios, however, we are not able to run model-free prediction algorithms until guaranteed convergence, which would require an infinite amount of episodes, and thus we cannot fully trust our estimates of the value function. We account for this by using ϵ-greedy policies, which keep our policies stochastic as compared to the deterministic policies produced by greedy maximization [34]:

$$\pi'_\epsilon(a|s) \leftarrow \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} & a = \arg\max_{a \in A} Q_\pi(s, a) \\ \frac{\epsilon}{|\mathcal{A}(s)|} & else \end{cases}$$

Where $Q_\pi(s, a)$ is our current estimate of the action value function of policy π. We can

now use either MC or TD prediction as our policy evaluation method in conjunction with ϵ-greedy policy improvement to yield a model-free control algorithm. Shown below is Sarsa(0), a control algorithm that uses TD(0) as its prediction method: Sarsa is what is known as an *on-policy* control algorithm, in the sense that the agent

Algorithm 2 Sarsa control for estimating $Q \approx q_*$, $\pi \approx \pi_*$ [39]

procedure SARSA(N, γ)
    $Q(s, a) \leftarrow 0 \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$
    $n \leftarrow 0$
    repeat
        Obtain initial state S
        $A \leftarrow action \sim \pi_\epsilon(a|S)$
        repeat
            Take action A, obtain R, S'
            $A' \leftarrow action' \sim \pi_\epsilon(a|S')$
            $Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A))$
            $S \leftarrow S'$
            $A \leftarrow A'$
            $\pi_\epsilon \leftarrow \epsilon - greedy(Q)$
        until *terminated*
        $n \leftarrow n + 1$
    until $n = N$
    $\pi \leftarrow greedy(Q)$
    Output Q, π
end procedure

is behaving with respect to policy π_ϵ and the control algorithm is learning $Q \approx q_{\pi_\epsilon}$ to guide its policy improvements. *Off-policy* control algorithms exist where we learn the action-value function of policy *different* from the one we are using to guide agent behavior. *Q-Learning* is a notable off-policy control algorithm that learns the action-value function of q_* while using π_ϵ to guide its behavior. Specifically, we can replace the update

$$Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A))$$

in Sarsa with

$$Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma \max_a Q(S', a) - Q(S, A)) \tag{17}$$

to yield the Q-Learning control algorithm.

1.3 Practical Reinforcement Learning

The control algorithms previously discussed (Sarsa and Q-Learning) explicitly stored the an estimate of the action-value function for every tuple $(s, a), s \in \mathcal{S}, a \in \mathcal{A}$. While this approach could be easily implemented for MDPs with small state and action spaces via a hash table, this approach is unsustainable for large MDPs. Instead, we must *approximate* the *estimate* of our action-value function using function approximators that can generalize over state-action pairs. Specifically, the function approximators are parameterized by θ – yielding $Q(s, a; \theta)$. We wish to find the parameters θ^* that minimizes a cost function defined between $Q(s, a; \theta)$ and $q_\pi(s, a)$.

MDPs may also have bounded but continuous state and/or action spaces. This makes the max operation in Eq. 17 intractable. In these scenarios we must turn to *policy gradient* methods, where instead of approximating the action-value function, we instead parameterize our policy π with a set of parameters θ – $\pi(a|s; \theta)$. With policy gradient methods we seek to find the optimal parameters θ^* that maximize the agent's cumulative reward. Furthermore, policy gradient methods are useful in finding solutions to MDPs whose optimal policies are stochastic, since value function based approaches would be deterministic.

Modern agents trained with reinforcement learning heavily rely upon deep learning techniques for function approximation. We first discuss the theory of control algorithms that use linear value-function approximators. We then overview policy gradient methods and conclude the background section with a significant discussion of deep reinforcement learning algorithms (and the deep learning techniques that

enable them).

1.3.1 Linear Value-Function Approximators

A linear value function approximator, $Q(s, a; \mathbf{w})$ aims to approximate the true value function $q_\pi(s, a)$ as closely as possible, where the notion of close is a metric that must be defined. If we define our metric to be the mean-squared error between the true value function and our approximator:

$$\mathcal{J}(\mathbf{w}) = \mathbb{E}_\pi[(q_\pi(S, A) - Q(S, A; \mathbf{w}))^2] \tag{18}$$

then we can determine the optimal direction to change our parameters $\mathbf{w}$ using *Stochastic Gradient Descent* [27], where we sample the gradient of the expectation. Specifically, since

$$Q(s, a; \mathbf{w}) = \phi(s, a)^T \mathbf{w}$$

where $\phi(s, a)$ is the feature vector that encodes the state-action pair (s, a), we can write [36]:

$$\nabla_{\mathbf{w}} \mathcal{J}(\mathbf{w}) = -2(q_\pi(S, A) - Q(S, A; \mathbf{w}))\nabla_{\mathbf{w}} Q(S, A; \mathbf{w})$$

$$= -2(q_\pi(S, A) - Q(S, A; \mathbf{w}))\phi(S, A)$$

To minimize $\mathcal{J}_{\mathbf{w}}$, we must change our parameters in the direction of the *negative* gradient:

$$\Delta\mathbf{w} = -\frac{\alpha}{2}\nabla_{\mathbf{w}} \mathcal{J}(\mathbf{w}) = \alpha(q_\pi(S, A) - Q(S, A; \mathbf{w}))\phi(S, A) \tag{19}$$

where α is a parameter that controls the step size of our parameter update. Since we do not actually have $q_\pi(s, a)$ (otherwise we would not be approximating it), we cannot actually calculate the gradient with this form of the equation. Instead, we use a *target* determined by our prediction algorithm (MC or TD). To illustrate, if we were using TD(0), we would substitute $q_\pi(s, a)$ with the TD-target, $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}; \mathbf{w})$, in

Eq. 19. This yields:

$$\Delta \mathbf{w} = -\frac{\alpha}{2}\nabla_{\mathbf{w}}\mathcal{J}(\mathbf{w}) = \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}; \mathbf{w}) - Q(S, A; \mathbf{w}))\phi(S, A)$$

While, in general, SGD can find the optimal parameters $\mathbf{w}$ in the case of linear function approximators, it may take many steps to do so. Using linear function approximators gives us the opportunity to solve directly for the parameters $\mathbf{w}$ that minimize the cost function $\mathcal{J}$. Since we expect our change in parameters to be zero at the minimum of the cost function, we can state [36]:

$$\mathbb{E}[\Delta \mathbf{w}] = 0$$

$$\sum \alpha(q_\pi(S, A) - Q(S, A; \mathbf{w}))\phi(S, A) = 0$$

$$\sum \phi(S, A)(q_\pi(S, A) - \phi(S, A)^T \mathbf{w}) = 0$$

$$\sum \phi(S, A) q_\pi(S, A) = \sum \phi(S, A)\phi(S, A)^T \mathbf{w}$$

$$\mathbf{w} = [\sum \phi(S, A)\phi(S, A)^T]^{-1} \sum \phi(S, A) q_\pi(S, A)$$

Solving for $\mathbf{w}$ directly in this manner is an $O(N^3)$ operation, where N is the length of the feature vector $\phi(S, A)$. Therefore, linear function approximators have the nice property of being able to directly approximate the target on each step of policy iteration.

1.3.2 Policy Gradient Methods

Policies built around value function estimators are intuitive because agents chose actions they thought had the highest action-value in a greedy fashion. Unfortunately, such policies have significant drawbacks. Value function estimators do not have great convergence properties (as we saw, we had to trade off variance of Monte-Carlo estimators for the bias of TD estimators to improve convergence). Furthermore, such

policies will be deterministic, due to the agent's greedy maximization of actions with respect to the action-value function. Deterministic policies cannot, inherently, perform well on tasks whose optimal policies are stochastic. For example, such an agent might choose to play *Rock* every single turn in the game of *Rock-Paper-Scissors* instead of the optimal stochastic policy of evenly playing *Rock*, *Paper*, and *Scissors* (and be easily exploited as well). Such policies are also infeasible in high-dimensional or continuous action spaces, where a maximization over actions would be extremely expensive. Lastly, small changes in value function estimates can drastically affect policies; such high variance in behavior is undesirable.

We can address some of these issues by attempting to learn a policy *directly*, rather than building them on top of learned value function estimates. If we consider a parameterized policy:

$$\pi(a|s;\theta)$$

our goal becomes to learn the optimal parameters θ^* that maximizes the performance of the agent's behavior, which is measured by the expected cumulative reward the agent receives. This objective can be characterized as follows [37] [39]:

$$J(\theta) = v_{\pi_\theta}(s_1)$$

$$J(\theta) = \int_{s\in\mathcal{S}} \rho^{\pi_\theta}(s) \int_{a\in\mathcal{A}} \pi_\theta(s,a) r(s,a) da\, ds \tag{20}$$

where

$$\rho^{\pi_\theta}(s') = \int_{s\in\mathcal{S}} \sum_{t=1}^{\infty} \gamma^{t-1} p_1(s) p(s \to s', t, \pi_\theta) ds \tag{21}$$

$$p(s_1 \to s_t, t, \pi_\theta) = \int_{\{s_1, s_2, \ldots s_{t-1}\}} \prod_{i=1}^{t-1} \int_{a\in\mathcal{A}} \mathcal{P}^a_{s_i s_{i+1}}$$

A movement in the parameters θ implies both a shift in the distribution of actions executed at each state as well as in the distribution of states experienced, as implied by Eq. 21. Both these distributions have significant impacts on the objective function,

as demonstrated in Eq. 20. While we can determine the effect of such a shift on the action distributions – since we have directly parameterized $\pi(a|s;\theta)$ – it is difficult for us to determine the change in the state distribution, since it is dependent on the environment's transition dynamics, $\mathcal{P}$, which is information our model-free agent is not privy to. Fortunately, however, the *Policy Gradient Theorem* [39] provides us an expression for $\nabla_\theta J(\theta)$ that does not involve a gradient of the state distribution [37]:

$$\nabla_\theta J(\theta) = \int_{s\in\mathcal{S}} \rho^{\pi_\theta}(s) \int_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s;\theta) q_\pi(s,a) da ds \tag{22}$$

We must be able to sample the gradient if we are to employ stochastic gradient descent (or an equivalent variant optimizer). We can rewrite Eq. 22 as follows:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho^{\pi_\theta}}[\int_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s;\theta) q_\pi(s,a) da] \tag{23}$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho^{\pi_\theta}, a\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s;\theta) q_\pi(s,a)] \tag{24}$$

Similar to Monte-Carlo value function estimation, we can use the return, G_t as an unbiased sample of $q_\pi(S_t, A_t)$; this yields the REINFORCE algorithm [41] [39] This

Algorithm 3 REINFORCE - Monte Carlo Policy Gradient

procedure REINFORCE(π_θ, N)
    Initialize θ
    $n \leftarrow 0$
    repeat
        Generate an episode $\mathcal{E} = \{S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T\}$ using π_θ
        for $t \in \{0, ..., T-1\}$ **do**
            $G \leftarrow$ return from step t
            $\theta \leftarrow \theta + \gamma^t G \nabla_\theta \log \pi(A_t|S_t;\theta)$
        end for
        $n \leftarrow n + 1$
    until $n = N$
end procedure

version of the policy gradient algorithm has high variance, however we can reduce the

variance by introducing a *baseline*, $b(s)$. We can rewrite Eq. 23 as follows:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho^{\pi_\theta}}[\int_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s;\theta)(q_\pi(s,a) - b(s))da] \tag{25}$$

Since

$$\mathbb{E}_{s\sim\rho^{\pi_\theta}}[-b(s)\int_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s;\theta)da] = 0$$

A good baseline that is often chosen in the literature is $b(s) = v_\pi(s)$. In practice, however, we must estimate $v_\pi(s)$ using a function approximator, $V(s;\mathbf{v})$. We then simply replace line 8 in Algorithm 3 with

$$\theta \leftarrow \theta + \gamma^t(G_t - V(S_t,\mathbf{v}))\nabla_\theta \log \pi(A_t|S_t;\theta)$$

We can learn the function approximator using a method such as TD-learning, as discussed in the previous section.

We can further reduce variance by employing *actor-critic* methods. These methods substitute a biased sample of $q_\pi(s,a)$ instead of using the unbiased sample, G_t. For example, if we use one-step actor critic methods, we substitute for G_t the *one-step return* (or the TD-target, as discussed previously):

$$\delta_t = R_{t+1} + \gamma V(S_{t+1};\mathbf{v}) - V(S_t;\mathbf{v})$$

the update to θ can now be written as:

$$\theta \leftarrow \theta + \alpha\delta_t\nabla_\theta \log \pi(A_t|S_t;\theta)$$

where α is the learning rate. The complete algorithm for the one-step actor critic is detailed below [39]

Algorithm 4 One-step Actor-Critic

procedure ActorCritic(π_θ, $V_{\mathbf{w}}$, $\alpha^{\mathbf{v}}$, α^θ, N) [39]
    Initialize θ, $\mathbf{v}$
    $n \leftarrow 0$
    repeat
        Obtain S, the first state of episode
        $I \leftarrow 1$
        while S is not terminal **do**
            Sample $A \sim \pi_\theta$
            Take action A, observe S', R
            $\delta \leftarrow R + \gamma V(S'; \mathbf{v}) - V(S, \mathbf{v})$
            $\mathbf{v} \leftarrow \mathbf{v} + \alpha^{\mathbf{v}} I \delta \nabla_{\mathbf{v}} V(S; \mathbf{v})$
            $\theta \leftarrow \theta + \alpha^\theta I \delta \nabla_\theta \log \pi(A|S; \theta)$
            $I \leftarrow \gamma I$
            $S \leftarrow S'$
        end while
        $n \leftarrow n + 1$
    until $n = N$
end procedure

1.4 Deep Reinforcement Learning

Instead of using linear functions to approximate value-functions, we can use deep neural networks. Linear function approximators need a good set of features $\phi(S, A)$ in order to accurately estimate the true value functions, but extracting such features often requires in-depth domain knowledge which can be prohibitive. Furthermore, such manually curated features may take a significant time to develop and may not be exhaustive [15]. Deep neural networks can circumvent the need for the in-depth domain knowledge required to perform feature extraction, since neural networks are often arranged in a hierarchical fashion that lends itself to progressively more abstract feature extraction [13]. We first overview deep learning techniques and then discuss reinforcement learning methods that have successfully employed deep neural networks to either approximate value functions and/or parameterize policies.

1.4.1 Deep Learning

1.4.1.1 Feedforward Neural Networks

The feedforward neural network, or multi-layer perceptron (MLP), is the most basic form of deep learning architecture. The neural network consists of an *input layer*, followed by one or more *hidden layers*, followed by an *output layer*. Information flows in one direction only: from the input layer to the output layer – hence the name feedforward. Neural networks are used to learn a function $\hat{y} = \hat{f}(x;\theta)$ that approximates a function $y = f(x)$ [3]

An example of a feedforward neural network is shown in Figure 4. The basic unit of any neural network is a *neuron*, depicted by the circles in Figure 4. In the feedforward case, a neuron takes a vector input, $\mathbf{x}$, and computes the value $g(\mathbf{w}^T\mathbf{x}+b) \in \mathbb{R}$, where $\mathbf{w}$ and b are parameters of the neuron and g is a non-linear *activation* function. The stacking of non-linear functions of the input provides our model the capability to learn more difficult functions than standard linear models. In the neural network of Figure 4, the first hidden layer consists of four neurons, each of which takes an input $\mathbf{x} \in \mathbb{R}^3$. The outputs of each of the four neurons are concatenated together to form the input passed to the second hidden layer, $\mathbf{x} \in \mathbb{R}^4$. We can write the overall operation of the first hidden layer as $\mathbf{y} = g_1(\mathbf{W}_1^T\mathbf{x} + \mathbf{b}_1)$, where $\mathbf{W}_1 \in \mathbb{R}^{3\times4}$, $\mathbf{b}_1 \in \mathbb{R}^4$, and $\mathbf{x} \in \mathbb{R}^3$. Since Figure 4 has 3 layers (excluding the input layer), the parameters of the neural network θ are $\{\mathbf{W}_i, \mathbf{b}_i\}_{i=1\ldots3}$. Thus, the output of the neural network, given the input layer, $\mathbf{x} \in \mathbb{R}^3$ is:

$$\mathbf{y} = g_3(\mathbf{W}_3^T(g_2(\mathbf{W}_2^T(g_1(\mathbf{W}_1^T\mathbf{x} + \mathbf{b}_1)) + \mathbf{b}_2)) + \mathbf{b}_3)$$

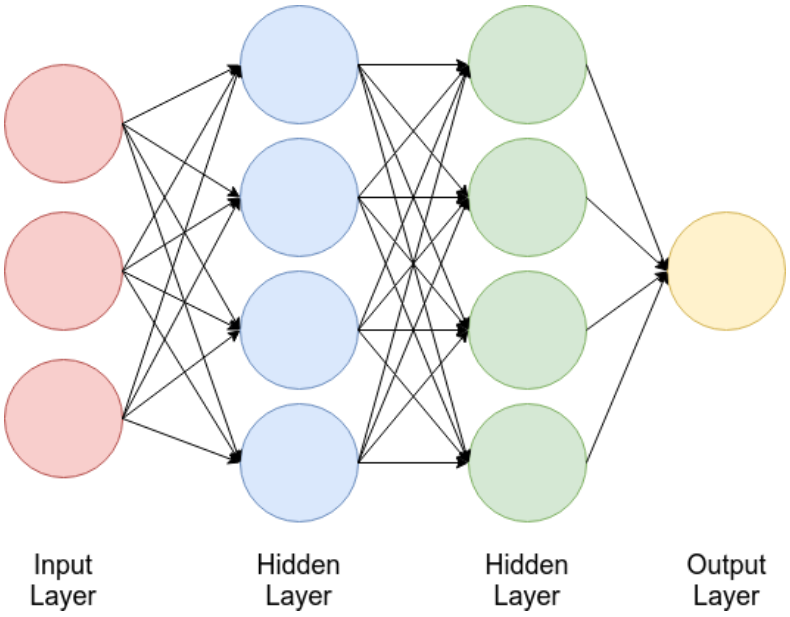

Figure 4: An example feedforward neural network with two hidden layers.

1.4.1.2 Output Layers, Cost Functions, and the Backpropagation Algorithm

Output layers of neural networks can vary depending on the task they are required to perform. We consider the two main learning tasks posed in Section 1.1, regression and classification.

- Regression - For regression we wish to closely approximate a function $f : \mathbb{R}^N \rightarrow \mathbb{R}$. Traditionally, we use an output layer consisting of one neuron with a linear activation function $g(z) = z$ for this case.

- Classification - For classification, we wish to predict the category $C \in \{1, ...k\}$ that an input $\mathbf{x} \in \mathbb{R}^N$ belongs to. For $k > 2$, our output layer consists of k neurons. We use the softmax function, $g(\mathbf{z})_i = \frac{\exp(\mathbf{z}_i)}{\sum_j \exp(\mathbf{z}_j)}$ to obtain probabilities $P(C = i|\mathbf{x})$ for $i = \{1...k\}$ [3]. For $k = 2$, our output layer consists of 1 neuron. We use the sigmoid function, $g(z) = \frac{1}{1+\exp(-z)}$ to obtain the probability $P(C = 2|\mathbf{z})$.

In order for the neural network to make predictions, the parameters θ must be learned using a dataset $\mathcal{D}$ consisting of labeled examples $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1...N}$. Neural networks employ the *Backpropagation algorithm* [28] to learn the parameters θ^* that minimizes a cost function $J(\theta)$. The Backpropagation algorithm is used to find the gradient $\nabla_\theta J(\theta)$ and take gradient descent steps until convergence:

$$\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta)$$

where α is the learning rate. Cost functions vary between tasks, but aim to capture the discrepancies between the function we wish to model and our estimation of that function. For regression, we can use the mean squared error:

$$J(\theta) = \mathbb{E}_{\mathbf{x},y\sim\mathcal{D}}[(y - \hat{f}(\mathbf{x};\theta))^2]$$

For classification, we can use the cross-entropy loss [3]:

$$J(\theta) = \mathbb{E}_{\mathbf{x},C\sim\mathcal{D}}[-\sum_{i=1}^{k} \mathbb{1}_{C=i} \log \hat{f}(\mathbf{x};\theta)_i]$$

$$\hat{f}(x;\theta)_i = P(C = i|x)$$

1.4.1.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a specialization of the neural network architecture that are used for processing grid-like input data, such as images [3]. CNNs have achieved state-of-the-art performance on many image recognition and object detection benchmarks, such as Mask R-CNN [7] which achieved state-of-the-art on MS COCO, a popular object detection competition.

A CNN is composed of layered *convolution* and *pooling* operations. The standard

definition of a convolution, from signal processing theory, is as follows:

$$y(t) = (x * w)(t) = \int x(a)w(t-a)da$$

This results in the common *flip and slide* interpretation of the convolution operation. CNNs employ convolution operations by setting x to be a multi-dimensional input, w to be an adaptable *kernel* of weights, and y to be the output of the operation. For example, a monochrome image can be represented by a 2-D matrix in $\mathbb{R}^{\text{height}\times\text{width}}$. We can represent the 2-D convolution of this image, X with a 2-D kernel k as follows:

$$Y(i,j) = (X * K)(i,j) = \sum_m \sum_n X(m,n)K(i-m, j-n)$$

Typically, however, we remove the *flip* component of *flip and slide* and are left with:

$$Y(i,j) = (X * K)(i,j) = \sum_m \sum_n X(m,n)K(i+m, j+n) \tag{26}$$

This is visualized in Figure 5. The convolution operations of CNNs, however, are not simply analogous to extending Eq. 26 to further dimensions. We will consider the three dimensional case, since most CNNs are built to process images. Images can be represented as a 3-D *tensor* $\in \mathbb{R}^{\text{height}\times\text{width}\times\text{channels}}$. For example, an 80×80 RGB image would be represented in $\mathbb{R}^{80\times80\times3}$. A convolutional layer processing an input tensor I of size $[C_I \times H_I \times W_I]$ will employ a *stack* of kernels $\{K_i\}_{i=1\ldots M}$, each of which will produce a *feature map*. Each of these kernels will be of the same size, $[H_K \times W_K]$, and consist of $H_K W_K$ weight vectors of length C_I, one at each location in the $H_K \times W_K$ kernel grid, for a combined size of $[C_I \times H_K \times W_K]$. The operation of kernel i on the input I can be written as:

$$Y_{i,j,k} = g(\sum_{l,m,n} I_{l,j+m-1,k+n-1}K_{i,l,m,n} + b_i) \tag{27}$$

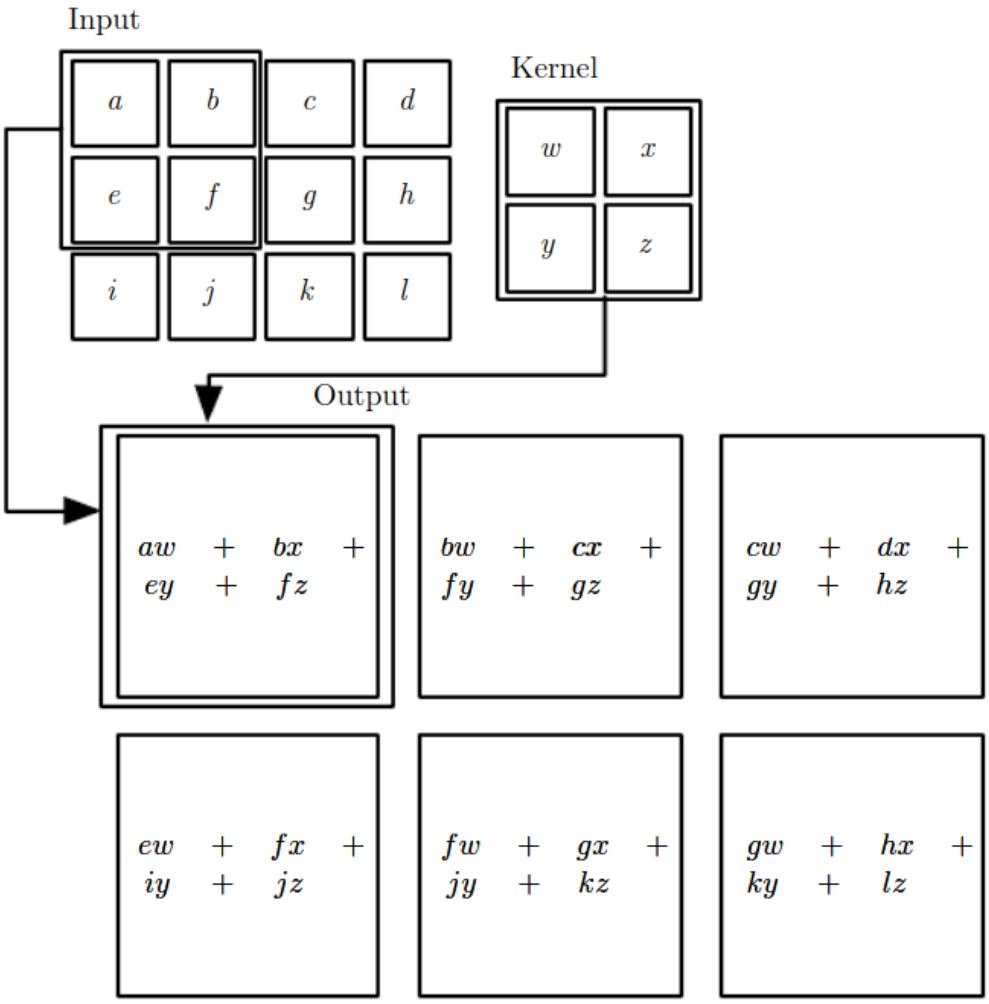

Figure 5: 2-D convolution without flipping and no activation function. The 2-D, [2 × 2] kernel block is slid over every [2 × 2] square block in the 2-D input. At each location, the kernel weights are multiplied by the respective value in the input, and the results are summed to yield a singular value at that location. The results at each location form the [2 × 3] output. [3]

where b_i is the bias parameter for kernel i and g is an activation function used to introduce nonlinearities. $H_K \times W_K$ is referred to as the *receptive field* of the kernel. This operation can be expensive for large inputs and kernel receptive fields; we can *downsample* our convolution by performing a *strided* convolution, where we slide our kernel by s units instead of by i unit in each direction. In this case, the operation of

kernel i on the input I can be written as:

$$Y_{i,j,k} = g(\sum_{l,m,n} I_{l,(j-1)s+m,(k-1)s+n} K_{i,l,m,n} + b_i) \tag{28}$$

Note that the summation over l, m, and n are only over indices where both I and K_i can be indexed validly. We can also *zero-pad* our input in both the height and width dimensions, such that the input size becomes $[C_I \times H_I + 2P_H \times W_I + 2P_W]$, where P_H is the amount of padding added to the height dimension, and P_W is the same but for the width dimension. Both zero-padding and strided convolutions help us control the size of the output of the convolutional operation, which is $[M \times H_O \times W_O]$ where

$$M = \text{number of kernels applied}$$

$$H_O = (H_I - H_K + 2P_H)/S_H + 1$$

$$W_O = (W_I - W_K + 2P_W)/S_W + 1$$

$$S = \text{stride length, in either the height or width dimensions}$$

Convolutional layers are used because they make use of two important ideas [3]

- Sparse Interactions - In a feedforward neural network, every neuron in every layer interacts with each output from the previous layer, resulting in a significant number of parameters needed to parameterize the model. In a convolutional network, however, since the kernel is typically *smaller* than the input, less weights are used, so the network is able to more efficiently model interactions between input variables.

- Parameter sharing - A convolution operation *slides* a kernel over an input to produce an output. Therefore, the parameters of the kernel are *reused* multiple times, each time the kernel is slid. The model therefore only has to learn one set of parameters that can be applied throughout all input locations, making

the model easier to learn, since far fewer parameters are learned. Furthermore, this allows the model to be *equivariant* to translation, which means that if an input is translated, the output is translated in the same fashion [3]

Pooling operations are usually applied between convolutional layers. Two main pooling operations exist: *max pooling* and *average pooling*. Any pooling operation performs a function within a rectangular area of its input. For example, max pooling applies the following operation:

$$Y_{i,j,k} = \max_{\substack{l=1+(j-1)h \ldots 1+jh, \\ m=1+(k-1)w \ldots 1+kw}} I_{i,l,m}$$

Pooling helps make CNNs *translationally invariant*, for small translations of input images, since pooling outputs are representatives of the inputs in each of their neighborhoods. Translational invariance can be an extremely useful characteristic for systems that must detect the existence of features, rather than the exact location of such features [3].

1.4.1.4 Recurrent Neural Networks

Recurrent neural networks (RNNs) are another type of specialized neural network architecture that excels at processing sequential data. RNNs leverage parameter sharing, just like CNNs, allowing RNNs to process variable length sequences and generalize across various positions. The output of an RNN is a temporally fixed function of previous outputs produced by the RNN:

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \theta) \tag{29}$$

In Eq. 29, $\mathbf{x}^{(t)}$ is an input to the RNN at time t, while $\mathbf{h}^{(t)}$ is the *state* of the network at time t. We can then apply a separate, temporally invariant function to $\mathbf{h}^{(t)}$ to yield predictions. For example, if we were trying to predict the next word given a sequence

of words of length L and a dictionary size of D, we would first apply the RNN L times to yield $\mathbf{h}^{(L)}$. A single-layer feedforward neural network with D units and a softmax activation function would then be used to predict the probability of the next word for each element of the dictionary. The recurrent functional form of Eq. 29 is what lends RNNs the ability to perform predictions on variable-length sequences, since it is specified in terms of a singular time-step transition and all inputs to the recurrent function are fixed in length ($\mathbf{h}^{(t)}$ and $\mathbf{x}^{(t)}$).

We can *unroll* an RNN by applying Eq. 29 t times. For example:

$$\mathbf{h}^{(2)} = f(f(\mathbf{h}^{(0)}, \mathbf{x}^{(1)}; \theta), \mathbf{x}^{(2)}; \theta)$$

Unrolling an RNN defines a computational graph from the beginning to the end of a sequence. Doing so allows us to then use backpropagation to update our network's parameters after a cost function is defined.

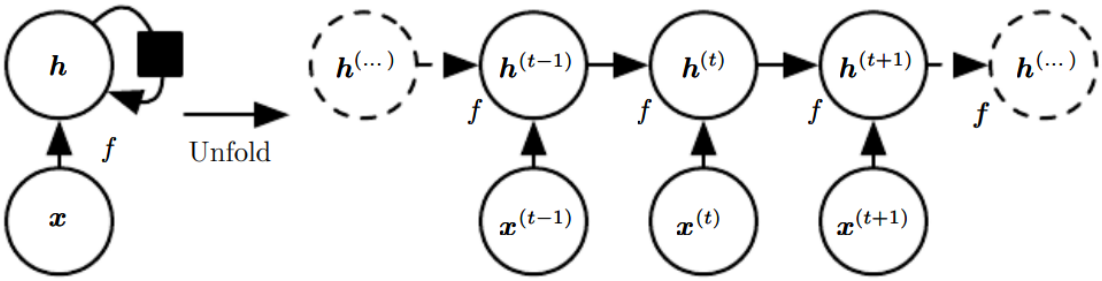

Figure 6: . A recurrent network unrolled (Eq. 29). The left image depicts the recurrent diagram, with the black box indicating the passage of a single time step. The right image depicts unrolling the recurrent diagram through time. The unrolled diagram forms a full computational graph from the beginning of time to the current timestep.[3]

We could also obtain predictions every timestep, as shown in Figure 7. In Figure 7, the following standard RNN update equations are applied:

$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)}$$

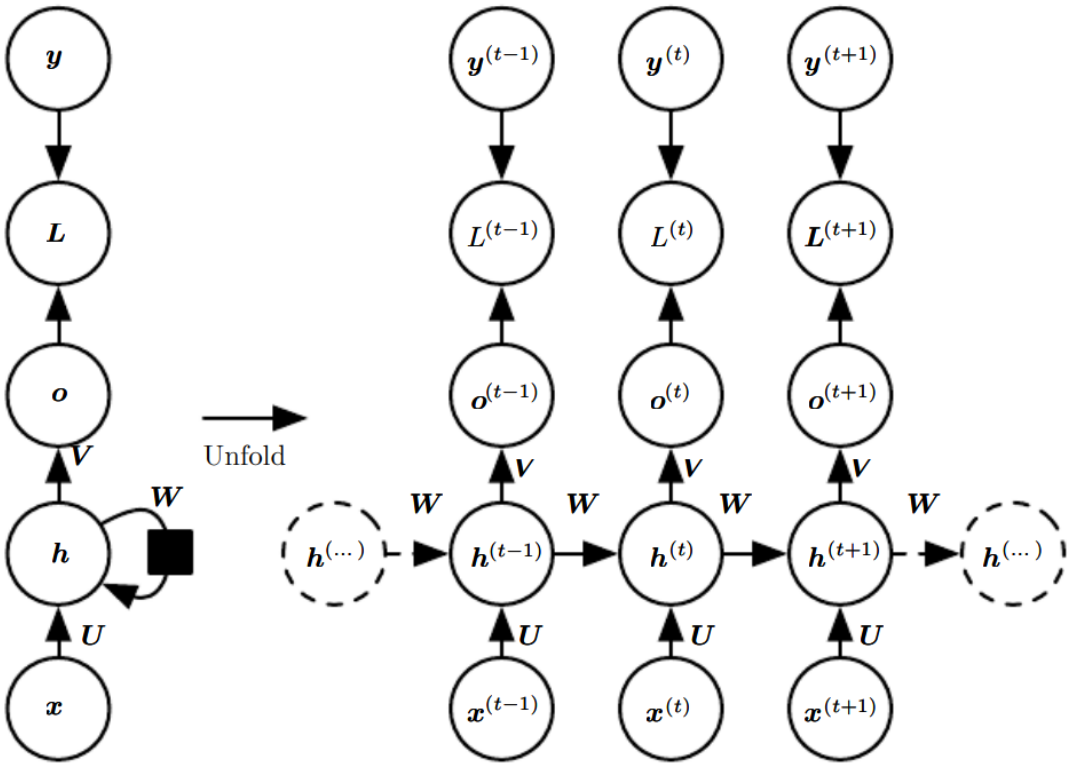

Figure 7: . The unrolled network that predicts a sequence of values $\mathbf{o}$ for an input sequence $\mathbf{x}$. At each timestep, a loss L is computed between a target, $\mathbf{y}$, and the output $\mathbf{o}$. The total loss of the network is $\frac{1}{\tau}\sum_{i=1}^{\tau} L^{(t)}$. Backpropagation is then used to find the parameters $\mathbf{W}$, $\mathbf{U}$, and $\mathbf{V}$ that minimize the expected loss of the network over a dataset of examples. [3]

$$\mathbf{h}^{(t)} = \tanh(\mathbf{a}^{(t)})$$

$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)}$$

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{O}^{(t)})$$

Backpropagating through an unrolled recurrent network is an expensive operation that costs $O(\tau)$ in both time and memory, where τ is the number of unrolled iterations. Furthermore, it is unparallelizable, since the output at each time step can only be computed after all previous timesteps have been passed through. Therefore, RNNs, on average, take longer to train than other neural network architectures.

Vanilla RNNs, as described thus far, face the well-known issues of *gradient vanishing* or *gradient explosion*, where gradients propagated across many timesteps either turn to zero or become exponentially larger [3]. The long short-term memory unit (LSTM) is one of the most common variations on the standard RNN that was introduced to handle these two problems. The LSTM network is a type of *gated RNN*, which attempts to create paths in the unfolded computational graph where gradients neither vanish nor explode. Gated units allow the network to accumulate and forget information over time. A LSTM cell is shown in Figure 8.

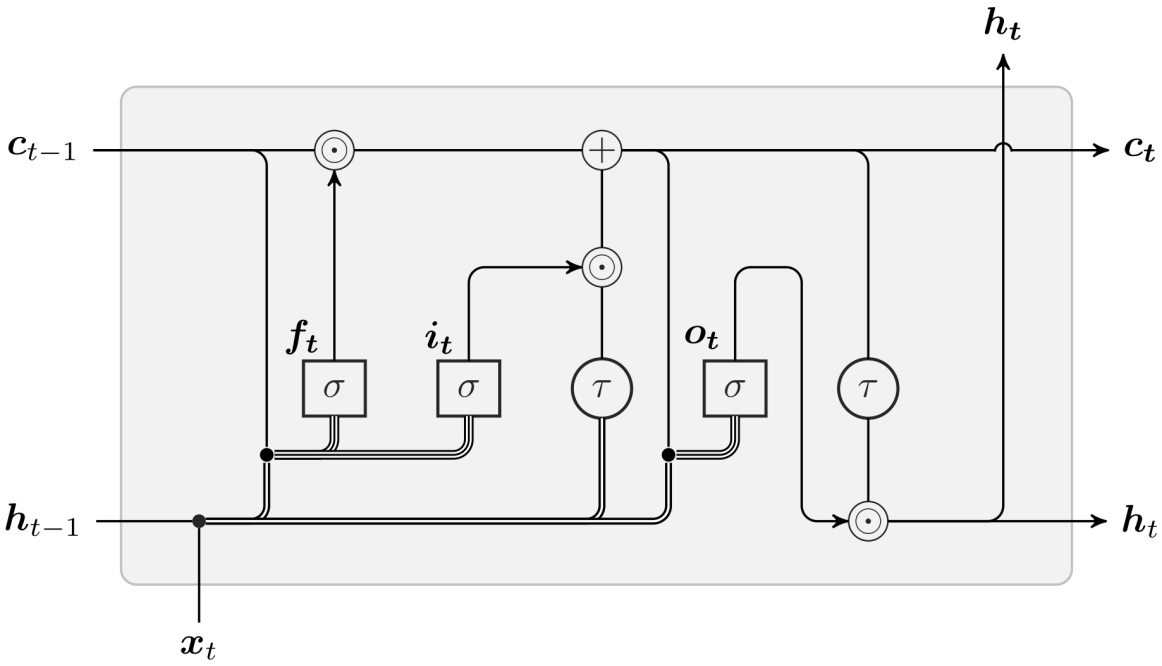

Figure 8: . An LSTM cell that has an inner recurrence on it [26]

The update equations are as follows [3]:

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f)$$

$$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i)$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o)$$

$$\mathbf{c}_t = \mathbf{f}_t \cdot \mathbf{c}_{t-1} + \mathbf{i}_t \cdot \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c)$$

$$\mathbf{h}_t = \tanh(\mathbf{c}_t) \cdot \mathbf{o}_t$$

Where $\mathbf{f}_t$ is the *forget* gate that controls how much of the cell state $\mathbf{c}_t$ gets passed over from one time step to another. $\mathbf{i}_t$ is the input gate that controls which cell state values we update. $\mathbf{o}_t$ is the output gate that controls which parts of the cell state we wish to output. LSTM networks have been successfully trained to learn both long-term and short-term dependencies, and are the most common type of RNN network employed when performing learning tasks on sequential data.

1.4.1.5 Regularization

As discussed in Section 1.1, we wish to make models that are able to generalize to new inputs. Neural networks are highly prone overfitting on training data, since the large number of parameters they have enables them to simply memorize the training set if they are not regularized carefully. There are many regularization schemes one can employ that either modify the cost function or perform augmentation on training data such that our models can achieve lower test error:

1. L^2 Regularization [12] [25] - L^2 regularization is a form of *parameter regularization* that adds the term $\frac{\lambda}{2}\sum_i w_i^2$ to the cost function of the neural network (note that the summation is only over the weights of our model, and not the biases). This term encourages weights to be closer to the origin and is also known as either *weight decay* or *ridge regression*. λ controls the strength of the regularization, the higher λ is, the stronger the regularization.

2. L^1 Regularization [25] - L^1 regularization is another form of parameter regularization that adds the term $\lambda \sum_i |\mathbf{w}_i|$ to the cost function of the neural network. L^1 regularization encourages the weights of our model to be *sparse*, such that the optimal values of some weights are zero. λ controls the strength of the regularization.

3. Dataset Augmentation [3] - For some tasks, it is simple to generate new dataset pairs that our model can train on. For example, if our model is aimed at classifying a picture of an animal, we can generate new data by applying slight blurs our dataset or by horizontally flipping each of the images in our dataset. Doing so will force our model to learn to classify blurry images and understand the symmetry of animals.

4. Early stopping [42] - The training error of neural networks can often decrease forever until it reaches zero – after the network has memorized its entire input dataset. During this process however, the validation error will begin to rise after the network has started to overfit on the training set. We can choose the best parameters of our model by selecting the point at which the model achieves the best validation error.

5. Dropout [38] - Dropout randomly samples a binary mask to apply to all input and hidden units. The binary mask zeros out the outputs of the respective units it covers. Dropout is usually applied with a probability $p = 0.5$, where p is the probability that the mask for each unit is "on". Dropout approximates the training of an ensemble of *subnetworks* that are constructed by removing nonoutput units from the original neural network.

1.4.1.6 Training Optimizations

Neural network training is often slow and requires many cycles, or *epochs* of sampling our dataset. We can improve convergence properties by utilizing these methods:

1. Minibatch Sampling [3] - Since our cost function is defined by:

$$J(\theta) = \mathbb{E}_{\mathbf{x},y\sim\hat{p}_{data}(\mathbf{x},y)}[L(f(\mathbf{x};\theta), y)]$$

finding the true gradient $\nabla_\theta J(\theta)$ requires performing a summation over the

entire training dataset, $\mathcal{D}$. We can instead, find the gradient of

$$\hat{J}(\theta) = \frac{1}{M} \sum_{i=1}^{M} L(f(\mathbf{x}_i; \theta), y_i), (\mathbf{x}_i, y_i) \sim \mathcal{D}$$

where M is our minibatch size. While Stochastic Gradient Descent (SGD), in expectation, samples the true gradient using a minibatch size of 1, larger batches are able to provide more accurate estimates of the gradient. Furthermore, larger batches are able to be processed in parallel, allowing the network to *see* the entire dataset faster.

2. Batch Normalization [9] - While training a neural network, the distribution of the inputs to each layer shifts, since the parameters of the network are being updated. This *internal covariate shift* requires us to carefully specify the learning rates and initial parameters of a model. *Batch Normalization*, introduced by Ioffe and Szegedy, 2015, is a normalization method that allows us to use larger learning rates and initialize parameters with less scrutiny. Batch normalization reduces internal covariate shift by normalizing inputs to each layer over the minibatch.

3. Rectified Linear Unit Activation [23] - As networks become deeper, gradients have an increased likelihood to vanish if sigmoid activations are used within hidden units. Instead, if the *rectified linear activation* is used, $g(x) = \max(0, x)$, gradients become less likely to vanish.

4. Variance Scaling Initializer [6] - As models become progressively deeper, it becomes more difficult to initialize the parameters of the model. Standard initialization schemes include initializing all weights from a $\mathcal{N}(0, v)$ distribution, where v is a hyperparameter. Such an initialization, however, may not avoid reducing or increasing the magnitude of the input by a significant amount as the input signal propagates through the network. The variance scaling initializer

initializes all weights from a $\mathcal{N}(0, \sqrt{(\frac{2}{n_l})})$, where n_l is the number of weights connecting an input to an output for layer l. This initialization scheme avoids the vanishing/exploding input problem and is able to allow deeper networks to converge.

5. Adam Optimizer [11] - Rather than applying the standard gradient descent update, $\theta \leftarrow \alpha \nabla_\theta J(\theta) + \theta$, one can apply an update with *momentum*. Adam is a momentum-based optimizer that works well with sparse gradients, invariant parameter updates, bounded step sizes, and automatic step size annealing. Adam computes estimates of the first and second moments of the gradient, which are used to compute the momentum-based updates.

1.4.2 Deep Learning and Reinforcement Learning

1.4.2.1 Deep Q-Networks

Deep Q-Networks (DQN) were first introduced in the landmark paper by Mnih et al. 2013, *Playing Atari with Deep Reinforcement Learning*, which ignited the field of Deep Reinforcement Learning. Prior to this work, it was not common to use neural networks since nonlinear function approximators are prone to causing instability or divergence in standard RL algorithms. DQN demonstrated usage of the same convolutional neural network architecture to train an agent to play *multiple* Atari 2600 games *without* feature engineering. The agent learned solely from RGB pixel inputs and the reward signal, as compared to linear learners which require heavy feature engineering. DQN was able to successfully mitigate instability issues and demonstrate state-of-the-art performance on a subset of the Atari 2600 games tested. Mnih et al. 2015 extended DQN with target networks to improve stability of the algorithm [20], and demonstrated improved results.

The two main innovations that DQN introduced were the usage of *experience replay* (originally introduced by [17]) and *target networks*. Experience replay uti-

lizes a *replay buffer* that stores *experiences*, $e_t = (s_t, a_t, r_t, s_{t+1})$ in a dataset, $\mathcal{D} = \{e_1, ..., e_N\}$. Q-learning updates are applied via minibatch updates obtained by sampling $\mathcal{D}$, rather than the online update discussed in earlier sections. Randomly sampling a minibatch of experiences removes the temporal correlations associated with online updates; furthermore, it allows the neural network to leverage experiences multiple times to perform gradient steps, bringing about greater data efficiency. DQN also utilizes target networks for bootstrapping, which reduces correlations between estimated action-values and their target values [20]. The full algorithm is detailed in Algorithm 5.

DQN specifically used a convolutional neural network to approximate the action-value function of the agent's policy, since CNNs are known to perform well for image-processing tasks. The agent's state, and thus the input to the neural network, were the last four frames of game history, which were cropped to the same 84×84 square region.

1.4.2.2 Double Deep Q-Networks (D-DQN)

Q-learning, in general, has been shown to produce overestimates of action-values [4]. This can be partially attributed to the construction of the Q-learning target, shown below:

$$\delta_t = R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta)$$

where the max operator both chooses an action and evaluates actions using the same values: $Q(\cdot; \theta)$, biasing the estimator towards overestimated values. If we, instead, learn two different estimates of the value function, $Q(S_{t+1}, a; \theta)$ and $Q(S_{t+1}, a; \theta')$, we can separate this process of selection and evaluation:

$$\delta_{t,\theta} = R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta); \theta')$$

Algorithm 5 Deep Q-Network [20]

procedure DQN(M, N, C, $\mathcal{G}$)
  Initialize replay memory $\mathcal{D}$
  Initialize action-value function approximator Q with random weights θ
  Initialize target action-value function approximator Q' with weights $\theta' = \theta$
  Initialize ϵ according to ϵ-annealing strategy $\mathcal{G}$
  for episode i:=1...M **do**
    Obtain S_1, the first state of episode
    while S_t is not terminal **do**
      Extract features $\phi_t \leftarrow \phi(S_t)$
      Select $a_t \sim \epsilon$-greedy policy π_ϵ w.r.t $Q(\cdot, \theta)$
      Perform a_t, observe s_{t+1}, r_t
      Extract features $\phi_{t+1} \leftarrow \phi(S_{t+1})$
      Store experience $e = (\phi_t, a_t, r_t, \phi_{t+1})$ into $\mathcal{D}$
      $\mathcal{B} \leftarrow$ A random minibatch of experiences $\{(\phi_j, a_j, r_j, \phi_{j+1})\}_N$ from $\mathcal{D}$
      for all experiences $\mathcal{E}_j = (\phi_j, a_j, r_j, \phi_{j+1}) \in \mathcal{B}$ **do**
        $$y_j \leftarrow \begin{cases} r_j & s_{j+1} \quad \text{is terminal} \\ r_j + \gamma \max_{a'} Q'(\phi_{j+1}, a'; \theta') & else \end{cases}$$
      end for
      $\theta \leftarrow \theta - \nabla_\theta \sum_{j=1}^{N} (y_j - Q(\phi_j, a_j; \theta))^2$
      Every C steps, $\theta' \leftarrow \theta$
    end while
    $\epsilon \leftarrow \epsilon'$ according to ϵ-annealing strategy $\mathcal{G}$
  end for
end procedure

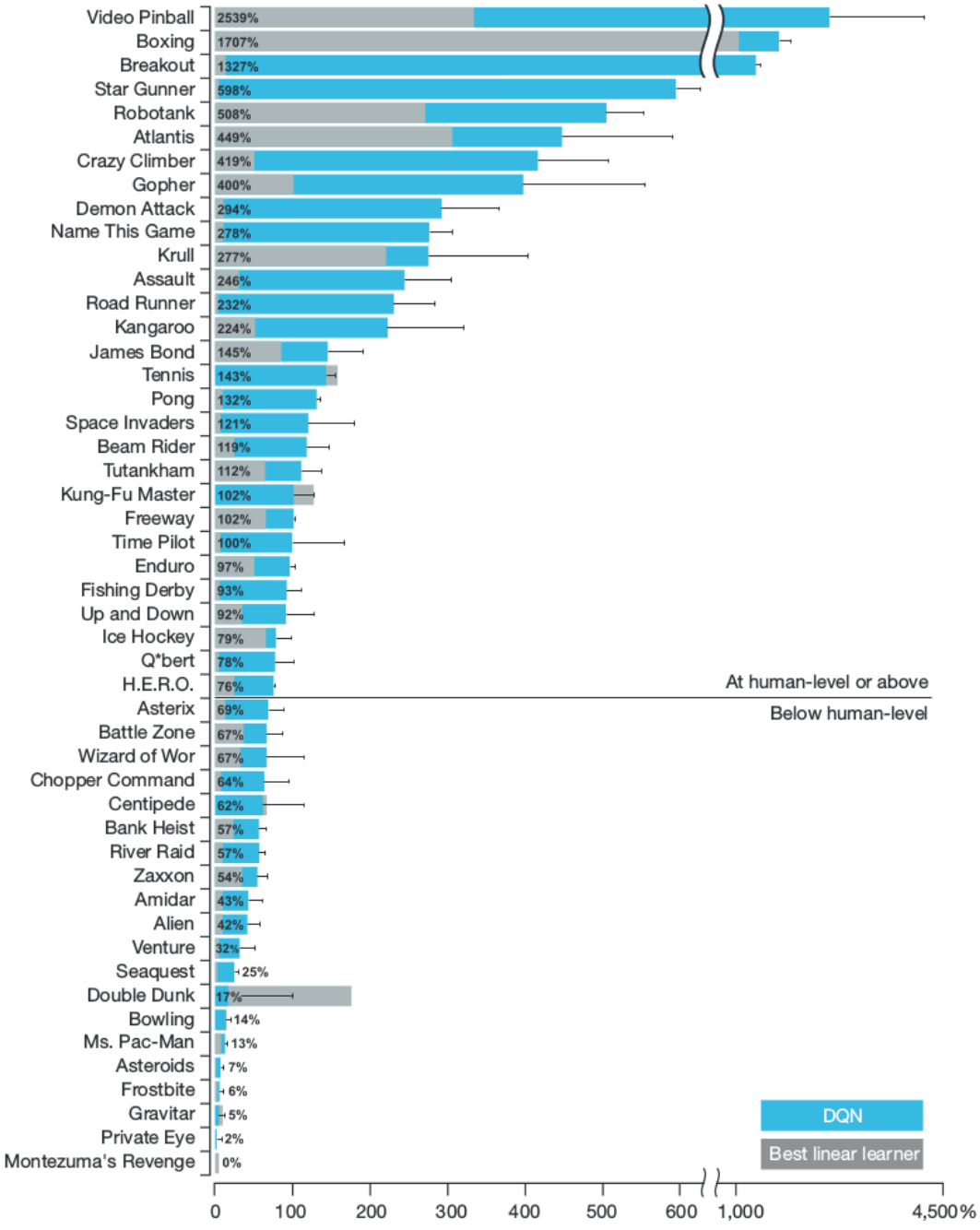

Figure 9: Performance of DQN vs using linear function approximators. DQN outperforms the best linear learner in all except three Atari games.[20]

Since DQN already makes use of two deep networks – an online and a target network, parameterized by θ and θ', respectively – we can leverage them for selection and

evaluation without having to introduce additional parameters [4]. Figures 10 and 11 depict the benefits of DDQN when training agents to play Atari games.

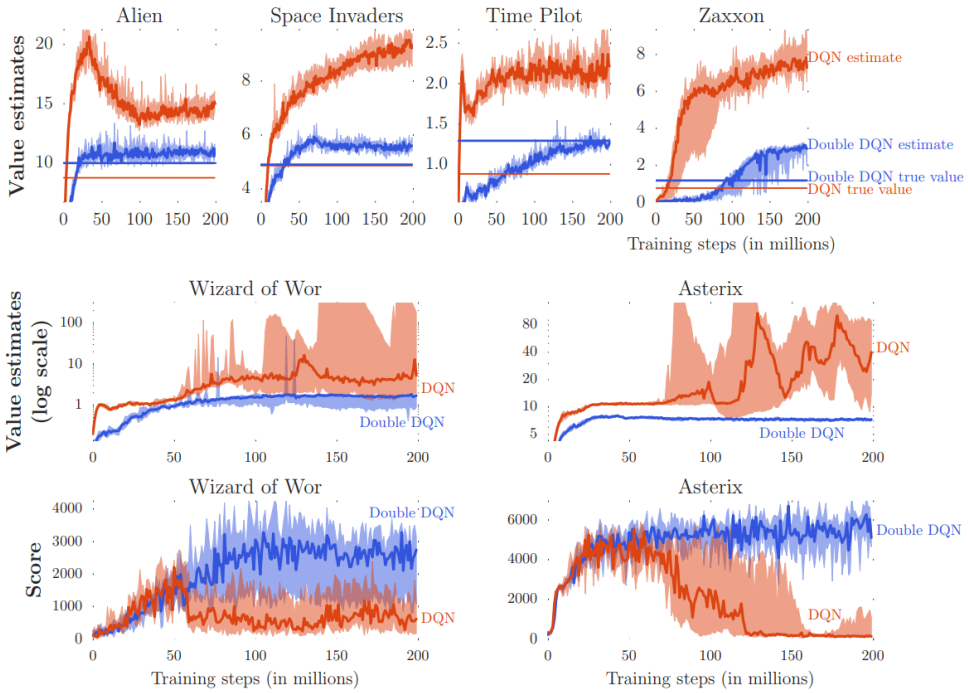

Figure 10: Learning curves of DDQN vs DQN on various Atari Games. DQN learning curves are shown in orange, while DDQN learning curves are shown in blue. The straight orange and blue lines show the actual cumulative reward achieved by the best DQN and DDQN agents, respectively, averaged over several episodes. If the learning algorithms were unbiased, their value function estimates would be equivalent to the straight lines at the end of training (right side of the plots). DDQN estimates are significantly less biased than DQN estimates. DDQN learning curves also exhibit greater stability than DQN's. Furthermore, DDQN's resultant policy performs better than DQN's in most games.[4]

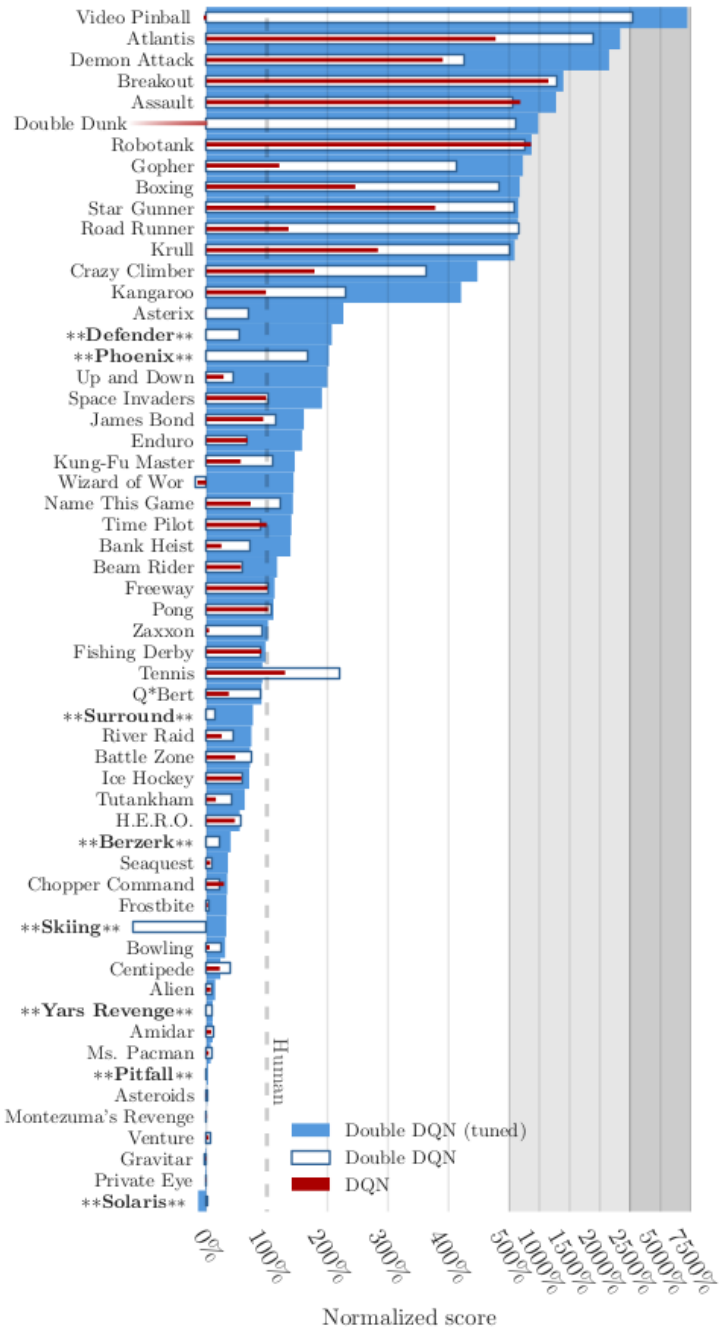

Figure 11: .Performance comparisons of DDQN vs DQN on numerous Atari games. DDQN was evaluated in the same manner as DQN. The white bars shows DDQN's performance with the same hyperparameters as DQN, while the blue bars indicate DDQN's performance with tuned hyperparameters. DDQN performs better than DQN in most games.[4]

1.4.2.3 Prioritized Experience Replay

While experience replay randomly samples experiences (s_t, a_t, r_t, s_{t+1}) uniformly from a replay buffer, prioritized experience replay samples experiences such that an agent is able to learn *faster*. The key idea which Schaul et al. 2015 used was to sample experiences based on their expected learning progress, as determined by the magnitude of their TD error [29]. While this can lead to issues – such as a loss of diversity in experiences the agent uses for learning and thus, an introduction of bias due to the change in distribution of experiences – stochastic prioritization and importance sampling can alleviate such problems. The stochastic sampling scheme introduced is detailed below:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

Where $P(i)$ is the probability of sampling experience i, p_i is the priority of experience i, and α is a parameter that controls how much we wish to prioritize experiences associated with higher TD errors. Two schemes are introduced to define priority:

1. $p_i = |\delta_i| + \epsilon$, δ_i is the TD-error for transition i, ϵ is a small, positive constant
2. $p_i = \frac{1}{rank(i)}$, $rank(i)$ = rank of transition i when the replay memory is sorted w.r.t $|\delta_i|$

Both of these schemes ensure that all experiences have a nonzero probability of being sampled and that an experience's sampling probability is a monotonic w.r.t its priority. Following this sampling method, however, would introduce bias in learning updates, since we would no longer be sampling the gradient of the objective function defined in Eq. 18. Therefore, we must *weight* an experience's update to the parameters by:

$$(\frac{1}{N} \cdot \frac{1}{P(i)})^\beta \cdot \frac{1}{\max_j w_j}$$

Double DQN with prioritized experience replay is detailed below:

Algorithm 6 Double DQN with Prioritized Experience Replay [29]

procedure DQNPRI(M, N, C, $\mathcal{G}$, K, γ, α, β, η)
    Initialize replay memory $\mathcal{D}$
    Initialize action-value function approximator Q with random weights θ
    Initialize target action-value function approximator Q' with weights $\theta' = \theta$
    Initialize ϵ according to ϵ-annealing strategy $\mathcal{G}$
    for episode i:=1...M **do**
        Obtain S_1, the first state of episode
        while S_t is not terminal **do**
            Extract features $\phi_t \leftarrow \phi(S_t)$
            Select $a_t \sim \epsilon$-greedy policy π_ϵ w.r.t $Q(\cdot, \theta)$
            Perform a_t, observe s_{t+1}, r_t
            Extract features $\phi_{t+1} \leftarrow \phi(S_{t+1})$
            Store experience $e = (\phi_t, a_t, r_t, \phi_{t+1})$ into $\mathcal{D}$
            if $t \equiv 0 \mod K$ **then**
                $\Delta \leftarrow 0$
                for j:=1...N **do**
                    Sample $e_j = (\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$ with $P(j) = \frac{p_j^\alpha}{\sum_i p_i^\alpha}$
                    $w_j \leftarrow \frac{(N \cdot P(j))^{-\beta}}{\max_i w_i}$
                    $y_j \leftarrow \begin{cases} r_j & s_{j+1} \text{ is terminal} \\ r_j + \gamma Q'(\phi_{j+1}, \arg\max_a Q(\phi_{j+1}, a; \theta); \theta') & else \end{cases}$
                    $\delta_j \leftarrow y_j - Q(\phi_j, a_j; \theta)$
                    $p_j \leftarrow |\delta_j|$
                    $\Delta \leftarrow \Delta + w_j \delta_j \nabla_\theta Q(\phi_j, a_j; \theta)$
                end for
                $\theta \leftarrow \theta + \eta \cdot \Delta$
                Every C steps, $\theta' \leftarrow \theta$
            end if
        end while
        $\epsilon \leftarrow \epsilon'$ according to ϵ-annealing strategy $\mathcal{G}$
    end for
end procedure

1.4.2.4 Dueling Networks

Previous algorithms discussed have mostly made improvements that were not specifically targeted towards the deep learning characteristics of the models, but rather the way reinforcement learning integrates with deep learning. Dueling networks, however, are a direct improvement towards the neural network action-value function approximator that is used in both DQN and Double DQN.

Previously, we discussed how policy gradient algorithms sample estimates of either $q_\pi(s,a)$ or $q_\pi(s,a) - b(s)$ to use in the gradient updates of equation 25. Furthermore, we discussed how a good baseline function used is $b(s) = v_\pi(s)$. The quantity

$$q_\pi(s,a) - v_\pi(s) = a_\pi(s,a) \tag{30}$$

is known as the *advantage function* which describes the relative benefit of performing action a when compared to the average return over all actions. In many states, the exact choice of action is almost inconsequential, while in other states it is of great importance. Therefore, it makes sense to estimate the advantage of a state-action pair rather than its action value. However, it is also essential to estimate the state-value function since it is necessary for bootstrapping [40].

Wang et al. 2016, introduced the dueling network architecture to estimate both the state-value and the advantage functions. The architecture introduced is shown in Figure 12. The architecture has two streams of information flow, one which provides an estimate of the value function, $V(s;\theta_v)$, and another which provides an estimate of the advantage function, $A(s,a;\theta_a)$. The two estimates are then combined to produce $Q(s,a)$, an estimate of the action-value function, which is used as the function approximator for the DQN or DDQN algorithms.

The estimates, however, cannot be combined as simply as:

$$Q(s,a) = V(s;\theta_v) + A(s,a;\theta_a)$$

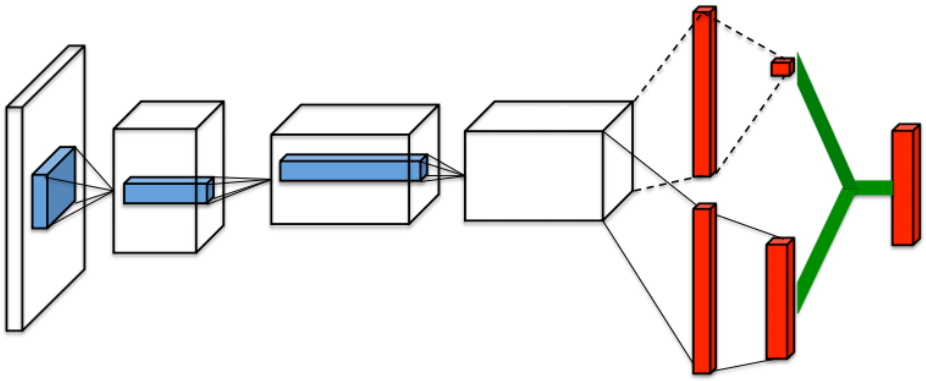

Figure 12: Dueling network architecture [40]

since opposite shifts in the function approximators by the same constant would still produce the same value:

$$V(s;\theta_v) + A(s,a;\theta_a) = (V(s,a;\theta_v) - k) + (A(s,a;\theta_a) + k)$$

such shifts would imply that the information being learned by the separate streams are not actually the state-value and advantage functions. Rather, if we consider the optimal deterministic policy π^* and the optimal action $a^* = \arg\max_a q_{\pi^*}(s,a)$, then $a_{\pi^*}(s,a^*) = 0$. Therefore, the estimates are combined as follows:

$$Q(s,a) = V(s;\theta_v) + (A(s,a;\theta_a) - \max_{a'} A(s,a';\theta_a))$$

With this separation of information, they were able to improve the results of DDQN on many Atari games.

1.4.2.5 Deep Recurrent Q-Networks (DRQN)

The DQN algorithm proposed by Mnih et al (2015) utilized a feature vector $\phi(s_t)$ composed of a stack of four 84×84 cropped images, each from the last four frames of history of the Atari game the agent was training on [21]. DQN thus forms poli-

cies based on states with limited history and will be unable to master games whose optimal policies require performing actions dependent on a history of more than four frames. In these scenarios, the state being supplied to the DQN no longer satisfies the *Markov assumption* we made earlier when discussing the general RL framework. The game, therefore, can no longer be represented by an MDP and must instead be represented by a *Partially Observable Markov Decision Process* (POMDP), which renders standard Q-learning useless. As discussed previously, RNNs have been able to form state representations that summarize sequences of arbitrary length. Such state representations would fit the Markov assumption more closely than those used by DQN. DRQN (Hausknecht & Stone, 2015) leverages RNNs to build its Q-function approximator so that the game can more closely fit into the standard MDP representation [5].

RNNs require *sequences* to be trained on – therefore, we must alter the way we sample experiences from the replay memory. In the DQN algorithm, experiences were uniformly sampled from the replay memory without taking into account the timestep at which the experience occurred at. Instead we must sample a batch of *sequential* experiences. We can either randomly sample full episodes or we could randomly sample a fixed-length sequence of experiences. Sampling full episodes allows us to set the RNN's initial state to zero and propagate an update state at each timestep, until the end of the episode. On the other hand, sampling fixed-length sequences require us to set the RNN's initial state to zero at a timestep that might not occur at the beginning of an episode. While the former sampling scheme is more intuitive, both sampling schemes were shown to yield similar agent performance [5]. Thus, the latter's computational simplicity lends itself to being the sampling choice in practice.

For the Atari games that DQN reported performance for, four frames of history was sufficient to guarantee that the game satisfied MDP assumptions. Therefore, Hausknecht & Stone 2015 introduced *Flickering Atari Games* to change the Atari environment from an MDP to a POMDP. DRQN modified Atari games such that

at every timestep, there was a probability $p = 0.5$ that the frame would be fully obscured. This modification required the agent to incorporate knowledge from a history of timesteps. The Q-function approximator consisted of convolutional layers, a single-layer LSTM, and a single fully connected layer with 18 hidden units (the number of available actions), and the agent was trained on sequences of 10 frames, through the BPTT algorithm. [5] demonstrated that the agent was able to successfully play the flickering version of *Pong*, a game that requires an understanding of object velocities (which is dependent on multiple timesteps of position information). Unlike the game environments that many RL algorithms are tested on, physical environments can rarely be fitted into the MDP framework. DRQN provides a potential avenue to help approximate POMDPs as MDPs to utilize Deep Q-Learning.

1.4.2.6 Generalized Advantage Estimation

Previous policy gradient methods discussed sampled the gradient

$$\nabla_\theta J(\theta) = \mathbb{E}_{s_{0:\infty}, a_{0:\infty}}[\sum_{t=0}^{\infty} A^{\pi,\gamma}(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t)]$$

where $A^{\pi,\gamma}$ is the discounted advantage function of the policy π with discount factor γ:

$$A^{\pi,\gamma} = q_\pi^\gamma(s_t, a_t) - v_\pi^\gamma(s_t)$$

$$q_\pi^\gamma(s_t, a_t) = \mathbb{E}_{s_{t+1:\infty}, a_{t+1:\infty}}[\sum_{l=0}^{\infty} \gamma^l r_{t+l}]$$

$$v_\pi^\gamma(s_t) = \mathbb{E}_{s_{t+1:\infty}, a_{t:\infty}}[\sum_{l=0}^{\infty} \gamma^l r_{t+l}]$$

If we consider the following estimator of $A^{\pi,\gamma}$:

$$\hat{A}_t = \delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t)$$

it is clear that it is unbiased for $V(s_{t+1}) = V^{\pi,\gamma}(s_{t+1})$:

$$\mathbb{E}_{s_{t+1}}[r_t + \gamma V^{\pi,\gamma}(s_{t+1}) - V^{\pi,\gamma}(s_t)$$

$$\mathbb{E}_{s_{t+1}}[Q^{\pi,\gamma}(s_t, a_t) - V^{\pi,\gamma}(s_t) = A^{\pi,\gamma}(s_t, a_t)$$

If $V(s_t)$ is an estimate of the state-value function, however, then we obtain a potentially biased estimator. We can lower the bias of our estimator by performing longer rollouts and considering $k - step$ estimates:

$$\hat{A_t^{(k)}} = \sum_{l=0}^{k-1} \gamma^l \delta_{t+l}^V = -V(s_t) + r_t + \gamma r_{t+1} + ... + \gamma^{k-1} r_{t+k-1} + \gamma^k V(s_{t+k})$$

as $k \to \infty$, the bias of $\hat{A_t^{(k)}} \to 0$. Similar to TD(λ), we can exponentially weigh the $k - step$ advantage estimates to trade-off between the bias and variance of our estimator. This results in the Generalized Advantage Estimate:

$$\hat{A_t^{GAE}} = (1-\lambda)(\hat{A_t^{(1)}} + \lambda \hat{A_t^{(2)}} + \lambda^2 \hat{A_t^{(3)}} + ...) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$$

Like TD(λ), a λ value closer to 0 has higher bias, while a lambda closer to 1 has higher variance.

1.4.2.7 Deterministic Policy Gradient

Our previous discussion of policy gradient algorithms was limited to the case of *stochastic* policies. Recalling Eq. 22, an integration over the action space is required to calculate the policy gradient, which results in the stochastic formulation of Eq. 24 where actions are sampled according to the current parameterized policy π_θ. In high dimensional scenarios, or in large spaces, a prohibitively high number of samples may be required in order to accurately follow the gradient of the objective function [37]. In this scenario, a *deterministic* policy, μ_θ can be used to more efficiently learn. It

is unclear, however, how such a policy can fit within the framework of Eq. 24. The Deterministic Policy Gradient algorithm (Silver et al. 2014 [37]) states that gradient of the objective function is as follows:

$$\nabla_\theta J(\theta) = \int_{s \in \mathcal{S}} \rho^\mu(s) \nabla_\theta \mu_\theta(s) \nabla_a q_\mu(s,a)|_{a=\mu_\theta(s)} ds \tag{31}$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{s \sim \rho^\mu}[\nabla_\theta \mu_\theta(s) \nabla_a q_\mu(s,a)|_{a=\mu_\theta(s)}] \tag{32}$$

where ρ^μ is defined analogously to ρ^π. Note the contrast with Eq. 24 – there is no second expectation with respect to the actions selected by the policy, since the policy is deterministic. Furthermore, we now take the gradient of the action-value function with respect to the action selected by the policy. We cannot, however, learn on-policy as suggested by Eq. 32, since we would not be exploring our action space adequately, resulting in convergence at a local optima. Therefore, our agent must learn using a *stochastic behavior policy*, β which allows the agent to explore while it learns the deterministic *actor policy*, μ. The gradient of the objective function can then be stated as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s \sim \rho^\beta}[\nabla_\theta \mu_\theta(s) \nabla_a q_\mu(s,a)|_{a=\mu_\theta(s)}] \tag{33}$$

In practice we approximate $q_\mu(s,a)$ through methods previously discussed, such as TD-learning or Q-learning.

1.4.2.8 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) combines the methods introduced in DPG [37] with the methods of DQN [21]. Specifically, they utilize the methods of replay buffer sampling and target networks that were introduced in DQN to stabilize the training of deterministic policy networks. They modify the abrupt target network

updates from DQN to soft updates such that the target networks slowly *track* the online networks. The full algorithm is detailed in Algorithm 7.

DDPG can also be extended, like DQN, with both Prioritized Experience Replay (for more efficient sampling of experiences) and Double-Q Learning (for obtaining better estimates from the critic).

1.4.2.9 Trust Region Policy Optimization

Previous policy gradient algorithms discussed did not enforce a guarantee on whether subsequent policies improved the objective function. The algorithms merely provided methods to sample the gradient of the objective function; therefore, the improvement of the policy is subject to high variance. Trust Region Policy Optimization (Schulman et al. 2015 [31] aims to provide greater assuredness that policy iteration steps result in actual improvement with regards to the objective function.

Letting

$$\eta(\pi) = \mathbb{E}_{s_0}[v_\pi(s_0)]$$

It has been shown that

$$\eta(\pi') = \eta(\pi) + \mathbb{E}_{s_0,a_0,\ldots\sim\pi'}[\sum_{t=0}^{\infty} \gamma^t A_\pi(s_t, a_t)]$$

We can rewrite $\eta(\pi')$ as follows:

$$\eta(\pi) + \sum_s \rho_{\pi'}(s) \sum_a \pi'(a|s) A_\pi(s,a)$$

where

$$\rho_\pi(s) = \sum_{t=0}^{\infty} \gamma^t P(s_t = s)$$

Therefore, if we are able to guarantee that $\sum_a \pi'(a|s) A_\pi(s,a) \geq 0 \quad \forall s$, then we can say with certainty that our policy has improved or stayed the same. However, since $A_\pi(s,a)$ is estimated, this cannot be guaranteed. Ensuring that the entire sum, $\sum_s \rho_{\pi'}(s) \sum_a \pi'(a|s) A_\pi(s,a) \geq 0$, requires knowledge of $\rho_{\pi'}$, which is often difficult to obtain. If, instead, we approximate $\eta(\pi')$ with:

$$L_\pi(\pi') = \eta(\pi) + \sum_s \rho_\pi(s) \sum_a \pi'(a|s) A_\pi(s,a)$$

We can use an estimate of the current policy's discounted state distribution, which is

Algorithm 7 Deep Deterministic Policy Gradient [16]

procedure DDPG(M, N, C, $\mathcal{G}$, K, γ, η_Q, η_μ, τ)
  Initialize replay memory $\mathcal{D}$
  Initialize critic $Q(s, a|\theta^Q)$ with random weights θ^Q
  Initialize actor $\mu(s|\theta^u)$
  Initialize target networks Q' and μ' with weights $\theta^{Q'} = \theta^Q$ and $\theta^{\mu'} = \theta^\mu$
  for episode i:=1...M **do**
    Initialize a random process $\mathcal{N}$ for exploration according to strategy $\mathcal{G}$.
    Obtain S_1, the first state of episode
    while S_t is not terminal **do**
      Extract features $\phi_t \leftarrow \phi(S_t)$
      Select $a_t = \mu(\phi_t|\theta^\mu) + \mathcal{N}_t$
      Perform a_t, observe s_{t+1}, r_t
      Extract features $\phi_{t+1} \leftarrow \phi(S_{t+1})$
      Store experience $e = (\phi_t, a_t, r_t, \phi_{t+1})$ into $\mathcal{D}$
      $\Delta^Q \leftarrow 0$
      $\Delta^\mu \leftarrow 0$
      for j:=1...N **do**
        Sample $e_j = (\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        $y_j \leftarrow \begin{cases} r_j & s_{j+1} \quad \text{is terminal} \\ r_j + \gamma Q'(\phi_{j+1}, \mu'(\phi_{j+1}|\theta^{\mu'})|\theta^{Q'}) & else \end{cases}$
        $\delta_j \leftarrow y_j - Q(\phi_j, a_j; \theta^Q)$
        $\Delta^Q \leftarrow \Delta^Q + w_j \delta_j \nabla_\theta Q(\phi_j, a_j; \theta^Q)$
        $\Delta^\mu \leftarrow \Delta^\mu + \nabla_a Q(\phi_j, a|\theta^Q)|_{a=\mu(s_j)} \nabla_{\theta^\mu} \mu(s|_{\theta^\mu})|_{s=s_j}$
      end for
      $\theta^Q \leftarrow \theta^Q + \eta_Q \cdot \Delta^Q$
      $\theta^\mu \leftarrow \theta^\mu + \eta_\mu \cdot \Delta^{\mu'}$
      $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
      $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
    end while
    $\epsilon \leftarrow \epsilon'$
  end for
end procedure

easier to obtain, to change our policy. $L_\pi(\pi')$ is a local, first-order approximation to $\eta(\pi')$ if we restrict ourselves to differentiable policies.

It has been shown that

$$\eta(\pi') \geq L_\pi(\pi') - \frac{4\epsilon\gamma}{(1-\gamma)^2} D_{KL}^{\max}(\pi, \pi')$$

where

$$D_{KL}^{\max}(\pi, \pi') = \max_s D_{KL}(\pi(s)||\pi'(s))$$

and D_{KL} is the KL-divergence between two distributions,

$$D_{KL}(P||Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} dx$$

We can then write, letting $C = \frac{4\epsilon\gamma}{(1-\gamma)^2}$

$$\eta(\pi') - \eta(\pi) \geq [L_\pi(\pi') - C D_{KL}^{\max}(\pi, \pi')] - [L_\pi(\pi) - C D_{KL}^{\max}(\pi, \pi)]$$

$$\eta(\pi') - \eta(\pi) \geq [L_\pi(\pi') - C D_{KL}^{\max}(\pi, \pi')] - [L_\pi(\pi)]$$

If we maximize the quantity $[L_\pi(\pi') - C D_{KL}^{\max}(\pi, \pi')]$, we are guaranteed to either improve or stagnate with respect to the objective function, since the lower bound of $\eta(\pi') - \eta(\pi)$ is guaranteed to be 0, if $\pi' = \pi$. Maximizing this quantity, theoretically, is a tradeoff between maximizing $L_\pi(\pi')$ and minimizing $D_{KL}^{\max}(\pi, \pi')$, i.e., finding the policy that maximizes the objective function while being close to the original policy. TRPO encapsulates policy update algorithms that solve the optimization problem:

$$\max_{\pi'} L_\pi(\pi') \quad \text{subject to} \quad D_{KL}^{\rho_\pi}(\pi, \pi') \leq \delta$$

where

$$D_{KL}^{\rho_\pi}(\pi, \pi') = \mathbb{E}_{s\sim\rho_\pi}[D_{KL}(\pi(s)||\pi'(s))]$$

1.4.2.10 Asynchronous Advantage Actor Critic (A3C)

All of the methods discussed thus far have relied on the use of a replay memory to train an agent. The replay memory serves to decorrelate updates to the agent and reduce the non-stationarity of the data [19]. Sampling past experiences, however, constrains us to off-policy learning methods, since the learned data is sampled from a distribution different from that derived from the agent's policy. Furthermore, using a replay memory results in greater computational and memory inefficiencies of learning algorithms. Mnih et al, 2016 introduced a framework for *asynchronous* training of agents. By training a set of parallel agents in parallel environments, one can obtain further decorrelated and stationary samples. Furthermore, this allows for more efficient training - the authors were able to train an agent that surpassed the previous state-of-the-art in half the time and using only a single, multi-core CPU (instead of a GPU). The most notable asynchronous algorithm developed in [19] is the Asynchronous Advantage Actor-Critic Algorithm, which fits the standard policy gradient algorithm (with a critic for baseline estimation) into the asynchronous framework. This algorithm is detailed in Algorithm 8.

1.4.2.11 Auxiliary Tasks

Many environments that agents are tasked to learn a policy in provide sparse reward signals. In such environments, it may be difficult for the agent to learn a policy quickly and to assign credit to actions. Mirowski et al. 2017 use *auxiliary* learning tasks to supplement the reward signal, allowing the agent to learn more efficiently [18]. Specifically, in [18], an agent is tasked with learning to navigate a complex maze, with the goal state providing a reward signal to the agent and "fruits" placed in the environment provide rewards to the agent for exploring its environment. These reward signals are sparse and thus [18] supplements the reward signal through the auxiliary task of *loop closure prediction* and *depth map inference*. Note that these tasks are beneficial for the agent if it wishes to learn how to navigate. For the first

Algorithm 8 Asynchronous Advantage Actor-Critic [19]

procedure A3C(T_{max}, t_{max}, θ, θ_v)
    // θ, θ_v are globally shared parameters
    // θ', θ'_v are thread-specific parameters
    $t \leftarrow 1$
    repeat
        $d\theta \leftarrow 0$, $d\theta_v \leftarrow 0$
        $\theta' \leftarrow \theta$, $\theta'_v \leftarrow \theta_v$
        $t_{start} = t$
        Observe state s_t
        repeat
            Perform $a_t \sim \pi(a_t|s_t;\theta')$
            Observe r_t, s_{t+1}
            $t \leftarrow t + 1$
            $T \leftarrow T + 1$
        until s_t is terminal, or $t - t_{start} = t_{max}$
        $\mathrm{R} = \begin{cases} 0 & s_t \quad \text{terminal} \\ V(s_t;\theta'_v) & s_t \quad \text{non-terminal} \end{cases}$
        for $i := t-1, ..., t_{start}$ **do**
            $R \leftarrow r_i + \gamma R$
            $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
            $d\theta_v \leftarrow d\theta_v + \nabla_{\theta'_v}(R - V(s_i;\theta'_v))^2$
        end for
        $\theta \leftarrow \theta + d\theta, \theta_v \leftarrow \theta_v + d\theta_v$
    until $T > T_{max}$
end procedure

task, the agent must predict whether it has navigated to a point that it has already visited before, while for the latter task, the agent must predict the depth of all pixels in the RGB image supplied to it.

Mirowski et al. 2017 train an agent to learn the auxiliary tasks by modifying the policy and value networks of a vanilla A3C agent. The modifications they perform are shown in Figure 13. They demonstrate that the modified A3C agent outperforms the vanilla A3C agent in a variety of navigation environments.

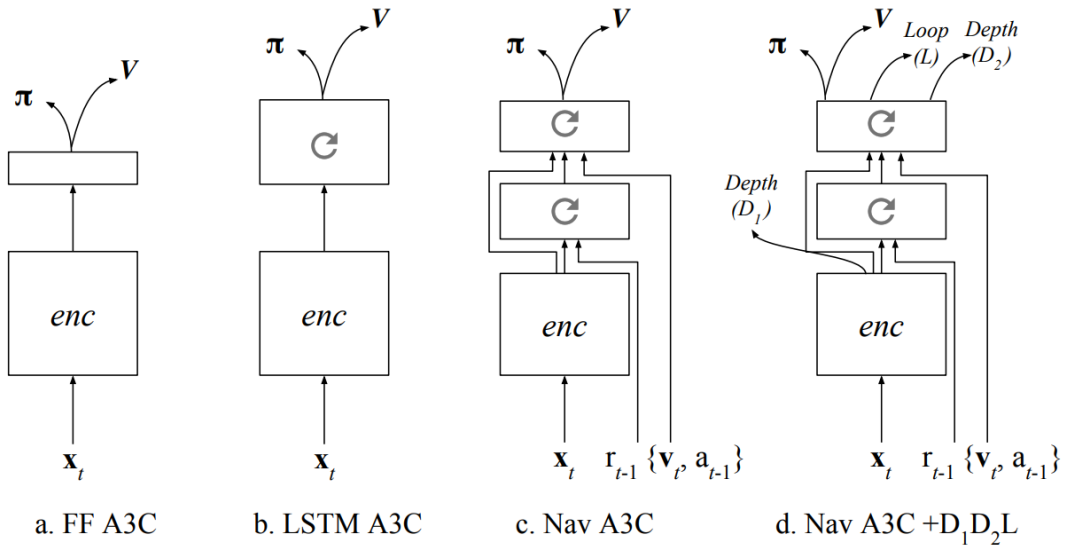

Figure 13: [18] uses the auxiliary tasks of loop closure prediction (L) and depth map inference (D_1, D_2) to provide denser training signals to their modified A3C agent shown in **d**. Backpropagation is used to minimize the losses associated with each task.

.

2 Problem Statement

We consider in this thesis the *portfolio management problem*, wherein we try to maximize our *cumulative wealth* after a certain time period. We primarily utilize the notation and development of Jiang et al. 2017 [10], supplemented by Necchi 2015

[24], and Li and Hoi 2013 [14].

Our agent starts with an *initial* portfolio value, p_0. At each time step t, it can reallocate a percentage of its total value, p_t, in each of m different assets, according to $\mathbf{b}_t$. This formalism necessitates that

$$\sum_{i=1}^{m} \mathbf{b}_{t,i} = 1 \tag{34}$$

where $\mathbf{b}_{t,i}$ is the fraction of the agent's total assets that are allocated towards asset i at time step t. Changes in the prices of the assets from time step t to time step $t+1$ result in a change, either positive or negative, in our agent's total holdings. We can write

$$\mathbf{x}_t = \mathbf{v}_t \oslash \mathbf{v}_{t-1} \tag{35}$$

where $\mathbf{x}_t$ is the *relative price vector* computed by taking the element-wise division of the *price vector*, $\mathbf{v}_t$, at time step t with respect to $t-1$.

$$\mathbf{v}_t \oslash \mathbf{v}_{t-1} = \Big(\frac{\mathbf{v}_{t,1}}{\mathbf{v}_{t-1,1}}, \frac{\mathbf{v}_{t,2}}{\mathbf{v}_{t-1,2}}, ..., \frac{\mathbf{v}_{t,m}}{\mathbf{v}_{t-1,m}}\Big)^T \tag{36}$$

Without considering any commission costs, we can state

$$p_t = p_{t-1}\mathbf{b}_{t-1}^T\mathbf{x}_t$$

our final, cumulative wealth after n periods is thus

$$p_n = p_0 \prod_{t=1}^{n+1} \mathbf{b}_{t-1}^T\mathbf{x}_t \tag{37}$$

where $\mathbf{b}_0$ is our initial portfolio allocation. The rate of return over period t is defined as

$$\rho_t = \frac{p_t}{p_{t-1}} - 1 = \mathbf{b}_{t-1}^T\mathbf{x}_t - 1 \tag{38}$$

and the logarithmic rate of return is defined as

$$r_t = \log \frac{p_t}{p_{t-1}} \tag{39}$$

We can then define the cumulative reward of the agent after n periods:

$$J = \sum_{t=1}^{n} r_t = \sum_{i=t}^{n} \log \mathbf{b}_{t-1}^T \mathbf{x}_t \tag{40}$$

Where the log returns of Eq. 39 have allowed us to convert the product in Eq. 37 to the sum in Eq. 40. This conversion is performed because reinforcement learning agents need an objective of the form of a sum of returns over timesteps, such as in Eq. 2. Note in both the portfolio allocation vector $(\mathbf{b}_{t,i})_{i=1\ldots m}^T$ and the price vector, $(\mathbf{v}_{t,i})_{i=1\ldots m}^T$, index $i = 1$ is reserved for cash. Therefore, $\mathbf{v}_{t,1} = 1$ for all t, and thus $\mathbf{x}_{t,1} = 1$.

The problem statement as thus formulated is both unrealistic and uninteresting. From a purely reinforcement learning perspective, the optimal action our agent can take at time step t is not dependent on steps before t, since the rewards our agent receives at step t is solely dependent on the action taken immediately before, $\mathbf{b}_{t-1}$, and the immediate dynamics of the environment, $\mathbf{x}_t$. Far-sightedness of our agent is thus useless, as individually maximizing rewards at each timestep will result in the greatest return (i.e., we can set $\gamma = 0$).

We can modify the current problem statement to include *commission* costs – on buying, selling, and holding. We shall first consider buying and selling commissions. Consider the price movements from time step $t-1$ to t and their effect on agent's portfolio $\mathbf{b}_{t-1}$. Right before the agent reallocates its assets to $\mathbf{b}_t$, its portfolio allocation is:

$$\mathbf{b}_t^{'} = \frac{\mathbf{b}_{t-1} \odot \mathbf{x}_t}{\mathbf{b}_{t-1}^T \mathbf{x}_t} \tag{41}$$

and its portfolio value is:

$$p_t^{'} = \mathbf{b}_{t-1}^T \mathbf{x}_t \tag{42}$$

After reallocating its assets to the portfolio allocation $\mathbf{b}_t$, commission costs consume a certain portion of the portfolio value $p_t^{'}$. We can summarize this with the *transaction factor* μ_t:

$$p_t = \mu_t p_t^{'} \tag{43}$$

The net wealth allocated towards each asset i just before reallocation is $\mathbf{b}_{t,i}^{'} \cdot p_t^{'}$. Analogously, after reallocation, it is $\mathbf{b}_{t,i} \cdot p_t$. If the net wealth before reallocation is greater than that after, then we have *sold* some amount of asset i, and have thus incurred a selling commission on that transaction. The net amount of cash obtained from selling is thus

$$(1 - c_s) \sum_{i=2}^{m} (p_t^{'} \mathbf{b}_{t,i}^{'} - p_t \mathbf{b}_{t,i})^+ \tag{44}$$

where c_s is the commission rate for selling, $0 \leq c_s < 1$, and $(x)^+ = \max(0, x)$. The total capital used to buy assets is thus

$$(1 - c_s) \sum_{i=2}^{m} (p_t^{'} \mathbf{b}_{t,i}^{'} - p_t \mathbf{b}_{t,i})^+ + p_t^{'} \mathbf{b}_{t,1}^{'} - p_t \mathbf{b}_{t,1} \tag{45}$$

after accounting for buying commission costs, the net amount of capital used to purchase new assets is

$$(1 - c_p) \Big[(1 - c_s) \sum_{i=2}^{m} (p_t^{'} \mathbf{b}_{t,i}^{'} - p_t \mathbf{b}_{t,i})^+ + p_t^{'} \mathbf{b}_{t,1}^{'} - p_t \mathbf{b}_{t,1} \Big] \tag{46}$$

where c_p is the commission rate for buying, $0 \leq c_p < 1$. This amount must equal the net non-cash assets purchased, thus

$$(1 - c_p) \Big[(1 - c_s) \sum_{i=2}^{m} (p_t^{'} \mathbf{b}_{t,i}^{'} - p_t \mathbf{b}_{t,i})^+ + p_t^{'} \mathbf{b}_{t,1}^{'} - p_t \mathbf{b}_{t,1} \Big] = \sum_{i=2}^{m} (p_t \mathbf{b}_{t,i} - p_t^{'} \mathbf{b}_{t,i}^{'})^+ \tag{47}$$

Substituting $p_t = \mu_t p_t^{'}$ and dividing by $p_t^{'}$ on both sides of Eq. 47, we can state

$$(1-c_p)\Big[(1-c_s)\sum_{i=2}^{m}(\mathbf{b}_{t,i}^{'}-\mu_t\mathbf{b}_{t,i})^{+}+\mathbf{b}_{t,1}^{'}-\mu_t\mathbf{b}_{t,1}\Big]=\sum_{i=2}^{m}(\mu_t\mathbf{b}_{t,i}-\mathbf{b}_{t,i}^{'})^{+} \tag{48}$$

We can use Eq. 48 to solve for μ_t. Specifically, we can use the following result proved by Jiang et al. 2017 [10], that sequence 49 converges to the true transaction factor μ_t, if $c_s = c_p = c$.

$$\Big\{\hat{\mu}_t^{(k)}|\hat{\mu}_t^{(0)}=\mu_{\odot} \quad \text{and} \quad \hat{\mu}_t^{(k)}=f\Big(\hat{\mu}_t^{(k-1)}\Big), k\in\mathbb{Z}^{+}\} \tag{49}$$

where

$$\mu_{\odot}=c\sum_{i=2}^{m}\Big|\mathbf{b}_{t,i}^{'}-\mathbf{b}_{t,i}\Big| \tag{50}$$

We can now expand Eq. 39 as

$$r_t=\log\frac{p_t}{p_{t-1}}=\log(\mu_t\mathbf{b}_{t-1}^{T}\mathbf{x}_t) \tag{51}$$

and reformulate the cumulative reward from Eq. 40 using commission costs:

$$J=\log\frac{p_f}{p_0}=\sum_{t=1}^{n}\log(\mu_t\mathbf{b}_{t-1}^{T}\mathbf{x}_t) \tag{52}$$

It is now apparent that the reward at time step t has recurrent dependencies – since μ_t is a function of $\mathbf{b}_{t-1}$, $\mathbf{b}_t$, and $\mathbf{x}_t$.

3 Related Work

We discuss selected pieces of significant related work. It is important to note that while policy-gradient based methods are the main methods discussed, other methods such as Q-learning have been applied.

3.0.0.1 Moody, et al. 1998

One of the first applications of reinforcement learning to the portfolio management problem was Moody, et al. 1998 [22]. Recognizing that the reward function of portfolio management is a recurrent function, they utilized recurrent learning updates to guide their agent. Specifically, for a utility function U_T capturing the economic benefit of the agent's actions, we can write

$$\frac{dU_T(\theta)}{d\theta} = \sum_{t=1}^{T} \frac{dU_T}{dR_t} \cdot \left\{ \frac{dR_t}{d\mathbf{b}_t} \frac{d\mathbf{b}_t}{d\theta} + \frac{dR_t}{d\mathbf{b}_{t-1}} \frac{d\mathbf{b}_{t-1}}{d\theta} \right\} \tag{53}$$

They used real-time recurrent learning (Williams and Zisper, 1989 [41]) to compute the derivatives of the agent's action with respect to the model parameters ($\frac{d\mathbf{b}_t}{d\theta}$).

In addition to using recurrent reinforcement learning (RRL), Moody et al. explored the use of different utility functions, specifically pure additive profits, power law utility functions with risk aversion, and differential Sharpe ratios. We will discuss the latter two as they are unique compared to the approach taken in this thesis.

Power law utility functions can represent different levels of risk sensitivity an investor may have [22]:

$$U_\nu(p_t) = \begin{cases} \frac{p_t^\nu}{\nu} & \nu \neq 0 \\ \log p_t & \nu = 0 \end{cases} \tag{54}$$

Risk aversion is then defined as

$$\mathcal{R}(p) = -\frac{d \log U'(p)}{d \log p} = 1 - \nu \tag{55}$$

Where $\mathcal{R}(p) = 0$ is risk-neutral (i.e., $\nu = 1$), and $\mathcal{R}(p)$ increasing implies heightened risk aversion.

Differential Sharpe ratios are a utility function that can allow an agent to maximize the Sharpe ratio when learning in an online fashion. The Sharpe ratio is a measure of risk-adjusted returns that modern portfolio managers use to quantify

their performance:

$$S_T = \frac{Average(R_t)}{StandardDeviation(R_t)} \tag{56}$$

$$S_T = \frac{A_T}{K_T(B_T - A_T^2)^{\frac{1}{2}}} \tag{57}$$

where

$$A_T = \frac{1}{T}\sum_{i=1}^{T} R_i$$

$$K_T = (\frac{T}{T-1})^{\frac{1}{2}}$$

$$B_T = \frac{1}{T}\sum_{i=1}^{T} R_i^2$$

For the differential Sharpe ratio, Moody et al. replace estimates of first and second moments by moving averages:

$$A_T = A_{T-1} + \eta(R_T - A_{T-1})$$

$$B_T = B_{T-1} + \eta(R_T^2 - B_{T-1})$$

Plugging in the above moving average estimates into Eq. 57, we obtain the differential Sharpe ratio:

$$D_t = \frac{dS_t}{d\eta} = \frac{B_{t-1}(R_t - A_{t-1}) - \frac{1}{2}A_{t-1}(R_t^2 - B_{t-1})}{(B_{t-1} - A_{t-1}^2)^{\frac{3}{2}}} \tag{58}$$

which can be used as a utility function U_t. It was shown in [22] that that training an agent to maximize the differential Sharpe ratio outperformed maximization of pure profit. Specifically, they note that as transaction costs increase, Sharpe ratio optimization outperforms that of pure profit when comparing the average cumulative wealth of various backtests on S&P 500 and 3-month T-bill time series.

3.0.0.2 Cumming, 2015 [2]

Cumming, 2015 targets the forex market specifically in their approach. They develop a novel feature extraction model for candlestick data, where $\mathbf{d}_t \in \mathbb{R}^4 = (open_t, high_t, low_t, close_t)^T$. Specifically, they define the *candlestick history* of length n to be

$$\mathbf{s}_h = (\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n)^T \in \mathbb{R}^{4n}$$

where $(\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n)$ are the past n candlesticks. They define m *center histories*, $\mathbf{c} \in \mathbb{R}^{m \times 4n}$ which are equally spaced through the dataset's candlestick histories. The center histories are used to compute a new feature vector

$$\phi_{gauss}(\mathbf{s}_h, \mathbf{c}) = (\exp -(\epsilon||\mathbf{s}_h - \mathbf{c}_1||)^2, \exp -(\epsilon||\mathbf{s}_h - \mathbf{c}_2||)^2, ..., \exp -(\epsilon||\mathbf{s}_h - \mathbf{c}_m||)^m)^T \in \mathbb{R}^m$$

where ϵ is a parameter that controls the width of each RBF term. Their agent uses a long/short MRP with sparse rewards produced only by closing out a position:

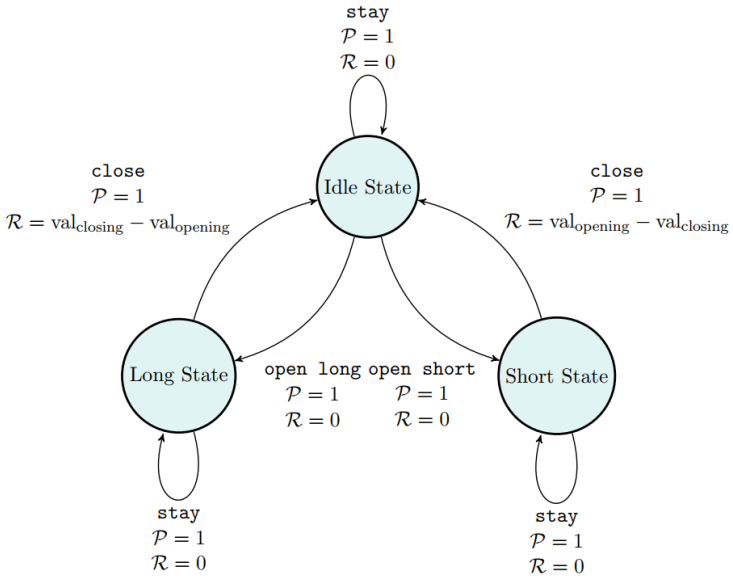

Figure 14: Cumming 2015's MRP model. The agent can either be in a long, short, or idle state. It only obtains a reward after closing out either a long or short state by returning to the idle state.[2]

Therefore, the agent in [2] is trained to maximize the cumulative profit realized by taking either long or short actions of equal magnitude. For each state, the agent receives different feature vectors based on the *unrealized profit and loss* it would receive for closing out its position. We define:

$$L_{pnl} = \text{value}_{\text{current}} - \text{value}_{\text{opening}}$$

$$S_{pnl} = \text{value}_{\text{opening}} - \text{value}_{\text{current}}$$

$$I_{pnl} = 0$$

The feature vector that the agent then uses to make decisions is:

$$\phi(\mathbf{s}_h, s) = \phi_{gauss}(\mathbf{s}_h, \mathbf{c}) \bigoplus \phi_{state}(s)$$

where $\bigoplus$ means the concatenation of both vectors, s is the agent's current state (either long, short, or idle), and

$$\phi_{state}(s) = \begin{cases} (S_{pnl}, 1, 0, 0)^T & s = \text{short state} \\ (I_{pnl}, 0, 1, 0)^T & s = \text{idle state} \\ (L_{pnl}, 0, 0, 1)^T & s = \text{long state} \end{cases}$$

Least-Squares Temporal Difference Learning (LSTD) and Least-Squares Policy Iteration (LSPI) is used by [2] instead of SGD to update their linear state-value function approximator of the form $\mathbf{w}^T \phi(\mathbf{s}_h, s)$. They utilize these methods instead of standard TD(λ) learning procedures because one can avoid tuning the learning rate parameter α of Eq. 19. Instead, one is just required to set the same λ parameter for bias-variance tradeoff as one would do in TD(λ).

Using one year's worth of data across 11 different currency pairs (around 350,000 candlesticks per pair), split into training and backtesting datasets, [2] achieves a

maximum annualized return of 0.0103% using LSPI. However, unlike the problem statement developed in Sec. 2, the action space of the agent is discrete, posing an easier and less general learning task. Furthermore, since positions can only be held or closed after opening them, there is no dependency of previous actions on the current commission costs experienced by the agent.

3.0.0.3 Necchi, 2016 [24]

Unlike both [22] and [2], Necchi, 2016 uses policy gradient methods to tackle the portfolio management problem developed in Section 2. Since portfolio management is a continuous process that does not have episodes, [24] modifies the reward function of Eq. 52 to the *average reward*:

$$J_{avg}(\theta) = \mathbb{E}_{\substack{S\sim\rho^{\theta} \\ a\sim\pi_{\theta}}}[\mathcal{R}(s,a)]$$

While one can use the policy gradient algorithms discussed earlier, such as Monte Carlo Policy Gradient (Algorithm 3), sampled histories can have great variability. To combat this issue, [24] uses Policy Gradient with Parameter-Based Exploration (PGPE). Instead of using a stochastic policy for exploration, PGPE explores the model parameter space θ directly, using a deterministic policy $\pi : \mathcal{S} \times \Theta \rightarrow \mathcal{A}$, drawing the policy parameters from a distribution, $\theta \sim p_{\zeta}$.

Similar to Cumming 2015, Necchi 2016 uses pure profit as the reward signal to guide the agent. However, Necchi does not use any feature extraction method for candlestick histories like Cumming. Instead, [24] uses the past n returns and the current portfolio allocation vector as their features:

$$\phi(s_t) = \{\mathbf{x}_{t-(n-1)}, \mathbf{x}_{t-(n-2)}, ...\mathbf{x}_t, \mathbf{b}_t^{'}\}$$

where $\mathbf{x}_t$ and $\mathbf{b}_t$ are defined as in Section 2. Using a simulated time series generated from an AR process, [24] trains a PGPE agent and a natural gradient variant,

NPGPE on seven thousand time samples and tests the agent on a two thousand time sample backtest. They also demonstrate their agent's accountability of commission costs by detailing how the agent's frequency of reallocation reduces significantly as commission costs are increased. Thus, the reinforcement learning approach to portfolio management successfully takes into account commission costs of the trading environment.

3.0.0.4 Jiang et al. 2017 [10]

Contrasting the previous works discussed, [10] uses deep learning methods to form the deterministic policy $\mu_\theta(s) = a$. They specifically target the cryptocurrency market, utilizing the same problem formulation as Section 2. The reward signal provided to the agent during training is the *average reward per timestep*

$$J = \frac{1}{n}\sum_{i=1}^{n} \log(\mu_t \mathbf{b}_{t-1}^T \mathbf{x}_t) \tag{59}$$

Jiang et al. 2017 also processes candlestick data, however they process the data differently from Cumming 2015. Using the notation of Section 2, the feature vector provided to the network at timestep t is $\{\mathbf{X}_t, \mathbf{b}_{t-1}\}$, where

$$\mathbf{X}_t = \left[\mathbf{V}_t, \mathbf{V}_t^{hi}, \mathbf{V}_t^{low}\right] \tag{60}$$

$$\mathbf{V}_t = \left[\mathbf{v}_{t-n+1} \oslash \mathbf{v}_t \middle| \mathbf{v}_{t-n+2} \oslash \mathbf{v}_t \middle| ... \middle| \mathbf{v}_{t-1} \oslash \mathbf{v}_t \middle| \mathbf{1}\right]$$

$$\mathbf{V}_t^{hi} = \left[\mathbf{v}_{t-n+1}^{hi} \oslash \mathbf{v}_t^{hi} \middle| \mathbf{v}_{t-n+2}^{hi} \oslash \mathbf{v}_t^{hi} \middle| ... \middle| \mathbf{v}_{t-1}^{hi} \oslash \mathbf{v}_t^{hi} \middle| \mathbf{1}\right]$$

$$\mathbf{V}_t^{low} = \left[\mathbf{v}_{t-n+1}^{low} \oslash \mathbf{v}_t^{low} \middle| \mathbf{v}_{t-n+2}^{low} \oslash \mathbf{v}_t^{low} \middle| ... \middle| \mathbf{v}_{t-1}^{low} \oslash \mathbf{v}_t^{low} \middle| \mathbf{1}\right]$$

and n is the window size of the model.

The main innovations of [10] are the Ensemble of Identical Independent Evaluators

(EIIE) policy network topology and the Portfolio Vector Memory (PVM) reformulation of the standard replay memory employed by DQN and similar deep reinforcement learning methods. A policy network can be considered an EIIE network if the same parameters θ are shared across evaluating different assets. Jiang et al. 2017 constructs EIIE networks of both CNN and RNN flavors, which we adapt and will discuss later. The parameter updates in [10] are executed as follows

$$\theta \leftarrow \theta + \alpha \nabla_\theta J_{[t_{b_1}, t_{b_2}]}(\pi_\theta)$$

where $[t_{b_1}, t_{b_2}]$ is a randomly sampled consecutive window of candlesticks from the cryptocurrency timeseries dataset. Note that although the paper terms this as a Deterministic Policy Gradient update, this does not follow the actual DPG algorithm proposed by [37]. Furthermore, the agent of Jiang et al. is maximally shortsighted, in the sense that $\gamma = 0$.

The PVM proposed by [10] facilities more efficient batch updates. The PVM is a stack of portfolio weight vectors arranged in chronological order. During training, the PVM is first initialized to uniform weights and each action the agent takes is stored in the PVM, in the respective slot. The agent reads weight vectors from the PVM to obtain $\mathbf{b}_{t-1}$, which its action is dependent on. Jiang et al. 2017 show that during the course of training, the contents of the PVM converge to that of the true actions the agent would have taken sequentially.

3.0.0.5 Zhang and Chen 2017 [43]

Zhang and Chen 2017 modify the problem statement of Jiang et al. 2017 such that the reward signal has no recurrent dependencies. Specifically, they consider an agent that begins each time period with all of its value in cash, reallocates to a diverse set of more risky assets, and then sells all assets back to cash at the end of the period. This problem formulation allows the agent's observation to solely be dependent on a

historical window of price data. The exact features that [43] provides to the agent is:

$$\phi(s_t) = \{\mathbf{x}_t, \mathbf{x}_{t-1}, \mathbf{x}_{t-2}, ..., \mathbf{x}_{t-n+1}\} \tag{61}$$

and the reward signal the agent is trained on is pure profit, averaged over the number of timesteps. Contrasting [10], [43] uses Deep Deterministic Policy Gradient (DDPG, discussed in section 1.4.2) to solve this problem statement. They show for a variety of window sizes and policy/critic network structures that DDPG is able to successfully solve their posed problem. Their results indicate that using a CNN for both the policy and critic network architectures with a window size of 3 performs the best in terms of cumulative wealth. Although they set $\gamma = 0.99$ in their model, maximal reward could have been achieved by setting $\gamma = 0$, since there is no dependence on future or previous timesteps in the reward signal.

4 Methods & Results

Deep Deterministic Policy Gradient (DDPG), along with TRPO and A3C, has been shown to be a state of the art continuous control learning method. We choose to utilize DDPG because of its greater interpretability, ease of implementation, and greater sample efficiency in stable environments [8]. Unlike Zhang and Chen 2017, we tackle the problem statement originally posed by Jiang et al. 2017. This problem statement poses a significantly more difficult learning task than that of [43] because of the multi-timestep dependency of the reward signal as discussed in section 2.

To more effectively handle this type of reward signal, we modify the original DDPG algorithm for n step learning. Furthermore, we incorporate auxiliary learning tasks into our model, as their effectiveness was demonstrated in [18]. Although the environment of [18] produced sparse rewards, we believe that auxiliary learning tasks can *bootstrap* the learning process. Such objectives can allow the networks to perform well on tasks that are necessary towards forming a good portfolio management policy.

4.0.0.1 $n - step$ DDPG

If we consider the problem statement formulation of Section 2, the reward observed by the agent at timestep t is a function of $(\mathbf{x}_t, \mathbf{b}_t, \mathbf{b}_{t-1})$. The reward consists of two components – the commission on the reallocation, which is dependent on $(\mathbf{b}_t, \mathbf{b}_{t-1})$ and the profit obtained on the changes in prices during the period, dependent on $(\mathbf{x}_t, \mathbf{b}_t)$. The original DDPG algorithm uses one step rollouts as detailed in Algorithm 7. One step rollouts, however, prevent the algorithm at any single timestep, from fully realizing the total effect of its action, as the commission and profit reward components are offset from each other by one timestep. We therefore modify the original DDPG algorithm to an $n - step$ formulation, detailed in Algorithm 9.

4.0.0.2 Actor and Critic Networks

We adapt the Ensemble of Independent Identical Evaluators (EIIE) from Jiang et al. 2017 [10] for use in both the actor and critic networks of DDPG. Specifically, we utilize two different paradigms for the models, an EIIE convolutional neural network (CNN) and an EIIE recurrent neural network (RNN). The state for the agent is a window of price returns, $\{\mathbf{x}_{t-n+1}, \mathbf{x}_{t-n+2}, ..., \mathbf{x}_t\}$ and the current portfolio allocation vector, $\mathbf{b}_t^{'}$. We choose to use $\mathbf{b}_t^{'}$ instead of $\mathbf{b}_{t-1}$ since it contains the most current information available. We depict the various structures of both the actor and critic networks in Figures 15 through 18.

4.0.0.3 Auxiliary Losses

In addition to $n - step$ DDPG, we modify Algorithm 9 to incorporate auxiliary losses, similar to Mirowski et al. 2017 [18]. While Mirowski et al. 2017 primarily uses auxiliary tasks to supplement the sparse rewards of their navigation environment, we use auxiliary tasks as a bootstrapping method, serving a similar role as the bias-variance tradeoff methods of TD(λ) methods. Although such learning tasks can aid the model in learning the overall portfolio management task, they can hamper the

overall learning task, causing early, sub-optimal convergence. We form two main auxiliary tasks – minimizing the net change in the agent's portfolio allocation vector $\mathbf{b}_t$ and performing a prediction of the price relative vector at the subsequent timestep, $\mathbf{x}_{t+1}$. The first of these auxiliary tasks can only be incorporated into the actor network and is implemented via a simple mean squared error loss term, $\frac{1}{N}\sum_{i=1}^{N}\sum_{i=1}^{m}(\mathbf{b}_{t,i}^{'} - \mathbf{b}_{t,i})^2$. The latter of these tasks can be implemented in both the actor and critic networks; the approach we take is detailed in Figure 19. The methodology for the CNN critic network and the RNN actor and critic networks follow an analogous approach. We use a mean squared error loss term, $\frac{1}{N}\sum_{i=1}^{N}\sum_{i=1}^{m}(\mathbf{x}_{t+1,i} - \hat{\mathbf{x}}_{t+1,i})^2$ to provide the learning signals for the predictive task. Thus we result in three auxiliary objectives: 1) applying the first auxiliary commission minimization task to the actor network, 2) applying the second auxiliary prediction task to the actor network, and 3) applying the second auxiliary prediction task to the critic network.

Algorithm 9 $n-step$ Deep Deterministic Policy Gradient

procedure DDPG(M, N, C, $\mathcal{G}$, K, S, γ, η_Q, η_μ, τ)
  Initialize replay memory $\mathcal{D}$
  Initialize critic $Q(s, a|\theta^Q)$ with random weights θ^Q
  Initialize actor $\mu(s|\theta^u)$
  Initialize target networks Q' and μ' with weights $\theta^{Q'} = \theta^Q$ and $\theta^{\mu'} = \theta^\mu$
  for episode i:=1...M **do**
    Initialize a random process $\mathcal{N}$ for exploration according to strategy $\mathcal{G}$.
    Initialize deque $\mathcal{Q}$
    Obtain s_1, the first state of episode
    Add s_1 to $\mathcal{Q}$
    for rollout step t :=1...$S-1$ **do**
      $s_t \leftarrow$ last element in $\mathcal{Q}$
      Extract features $\phi_t \leftarrow \phi(s_t)$
      Select $a_t = \mu(\phi_t|\theta^\mu) + \mathcal{N}_t$
      Perform a_t, observe s_{t+1}, r_t
      Add a_t, r_t, s_{t+1} to $\mathcal{Q}$
    end for
    while s_t = last element in $\mathcal{Q}$ is not terminal **do**
      Extract features $\phi_t \leftarrow \phi(s_t)$
      Select $a_t = \mu(\phi_t|\theta^\mu) + \mathcal{N}_t$
      Perform a_t, observe s_{t+1}, r_t
      Add a_t, r_t, s_{t+1} to $\mathcal{Q}$
      Store a copy of $\mathcal{Q}$ into $\mathcal{D}$
      $\Delta^Q \leftarrow 0$
      $\Delta^\mu \leftarrow 0$
      for j:=1...N **do**
        Sample rollout $\mathcal{Q}_e$ from $\mathcal{D}$
        $s_f \leftarrow$ first state in $\mathcal{Q}_e$
        $a_f \leftarrow$ first action in $\mathcal{Q}_e$
        $s_l \leftarrow$ last state in $\mathcal{Q}_e$
        $r_e \leftarrow$ sequence of all rewards in $\mathcal{Q}_e$
        $y_j \leftarrow \begin{cases} \sum_{i=1}^{S} \gamma^{i-1} r_{e,i} & s_l \quad \text{is terminal} \\ \sum_{i=1}^{S} \gamma^{i-1} r_{e,i} + \gamma^S Q'(\phi(s_l), \mu'(\phi(s_l)|\theta^{\mu'}); \theta^{Q'}) & else \end{cases}$
        $\delta_j \leftarrow y_j - Q(\phi(s_f), a_f; \theta^Q)$
        $\Delta^Q \leftarrow \Delta^Q + \delta_j \nabla_\theta Q(\phi(s_f), a_f; \theta^Q)$
        $\Delta^\mu \leftarrow \Delta^\mu + \nabla_a Q(\phi(s_f), a|\theta^Q)|_{a=\mu(\phi(s_f))} \nabla_{\theta^\mu} \mu(s|_{\theta^\mu})|_{s=\phi(s_f)}$
      end for
      $\theta^Q \leftarrow \theta^Q + \eta_Q \cdot \Delta^Q$
      $\theta^\mu \leftarrow \theta^\mu + \eta_\mu \cdot \Delta^{\mu'}$
      $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
      $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
    end while
    $\epsilon \leftarrow \epsilon'$
  end for
end procedure

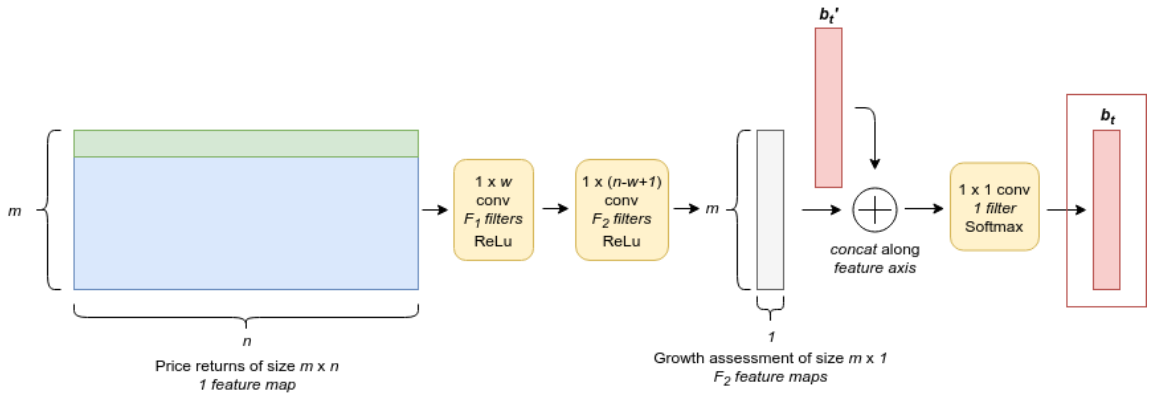

Figure 15: CNN actor network using the EIIE paradigm. Since the convolutional layers employ kernels of size $1 \times k$, the weights of each layer are shared between all asset classes. Therefore, the model cannot explicitly memorize features corresponding to individual asset classes. Furthermore, the EIIE paradigm prevents the network from memorizing which asset performs the best on the training set, allowing it to learn general features solely based on a history of price returns [10]. Two layers are used to first extract a feature vector based purely on the history of price relative vectors. We then incorporate the current portfolio allocation vector $\mathbf{b}_t^{'}$ by concatenating it along the feature axis. A 1×1 convolution is used to incorporate the current portfolio allocation vector into the agent's actions

.

4.0.0.4 Experimental Methodology

We demonstrate that DDPG and our variant $n-step$ DDPG can be used to tackle the portfolio management problem. Five years of S&P 500 data sourced from Kaggle are used, which amounts to 1825 daily candlesticks, starting from 2012-08-13 and ending at 2017-08-11. We specifically use the following tickers: AAPL, ATVI, CMCSA, COST, CSX, DISH, EA, EBAY, FB, GOOGL, HAS, ILMN, INTC, MAR, REGN, SBUX, selected because [43] curated the dataset. We modified two code repositories to obtain our results: Zhang et al's and OpenAI baseline's DDPG repositories, which can be found at `https://github.com/vermouth1992/drl-portfolio-management` and `https://github.com/openai/baselines/tree/master/baselines/ddpg` respectively. All models are trained and tested using a 60/40 train-test split.

Our agent is provided, at each timestep, a normalized version of the history of a

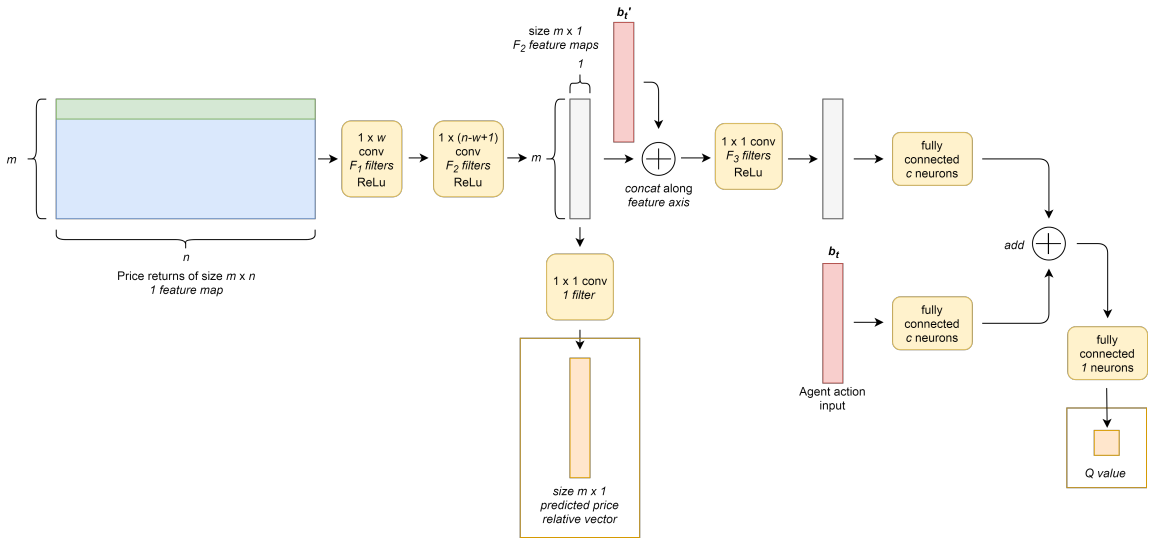

Figure 16: The CNN critic network uses the CNN actor network as its base. However, we no longer restrict the result of the 1×1 convolution after the feature concatenation to one feature map. Furthermore, we combine state and action information using fully connected networks. Both the state and action at timestep t are passed through fully connected networks of equal sizes. The results are added and then passed through a final fully connected layer of 1 neuron to result in the Q-value approximation. This idea of combining state and action information through adding is taken from the original DDPG paper [16].

.

certain window length, $f(\{\mathbf{x}_{t-n+1}, \mathbf{x}_{t-n+2}, ..., \mathbf{x}_t\})$, where $f(\mathbf{x}) = 100 * (\mathbf{x} - 1)$. This observation is used to perform the action $\mathbf{b}_t$. Our agent uses the average reward per time step, following Eq. 59, however we perform reward scaling by a factor of 1000. We use an Ornstein-Uhlenbeck process for exploration, inspired by the original DDPG paper [16]:

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t$$

with $\theta = 0.50$, $\sigma = 0.2$, $dt = 0.01$, $x_0 = 0$. Perturbed actions are re-clipped and normalized such that $\sum_i \mathbf{b}_{t,i} = 1$ and $\mathbf{b}_{t,i} \in [0, 1] \quad \forall i$.

For both the actor and critic CNN architectures, our first convolutional layer employs a 1×3 kernel, while the second kernel is dynamically sized based on the history length of the agent's observation, as detailed in Figure 15. Each of these

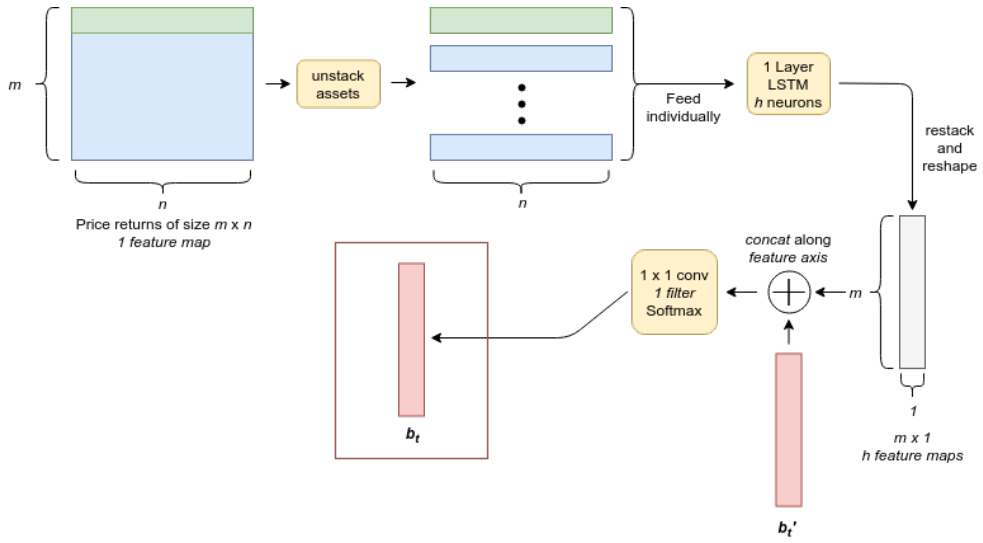

Figure 17: RNN actor network using the EIIE paradigm. The original price input features are unstacked along the asset axis and individually fed through a shared LSTM layer. The resulting output is restacked and reshaped to form a $m \times 1 \times h$ tensor. Similarly to the CNN actor, the network incorporates the current portfolio allocation vector $\mathbf{b}'_t$ by concatenating it with the output of the LSTM along the feature dimension. A 1×1 convolutional layer with a softmax activation results in the action the agent performs at timestep t.

.

layers uses 32 filters. For the LSTM architecture, we use a LSTM layer with 32 neurons for both the actor and critic. The final convolutional layer, using a 1×1 kernel, has 1 filter for both the actor and critic architectures. For the critic, state and action information are passed through two separate fully connected layers, as detailed in Figures 16 and 18, with 64 neurons each.

4.0.0.5 Experimental Results

We show the experimental results of various models on both the training and test sets. For all results shown, a batch size of 64 is used for each learning iteration. The actor and critic networks use the Adam Optimizer with learning rates of 0.0001 and 0.001, respectively. $\tau = 0.001$ is used for both networks. Shown in Figures 20 and 21 are the training and testing performance, respectively, of our $n - step$ DDPG

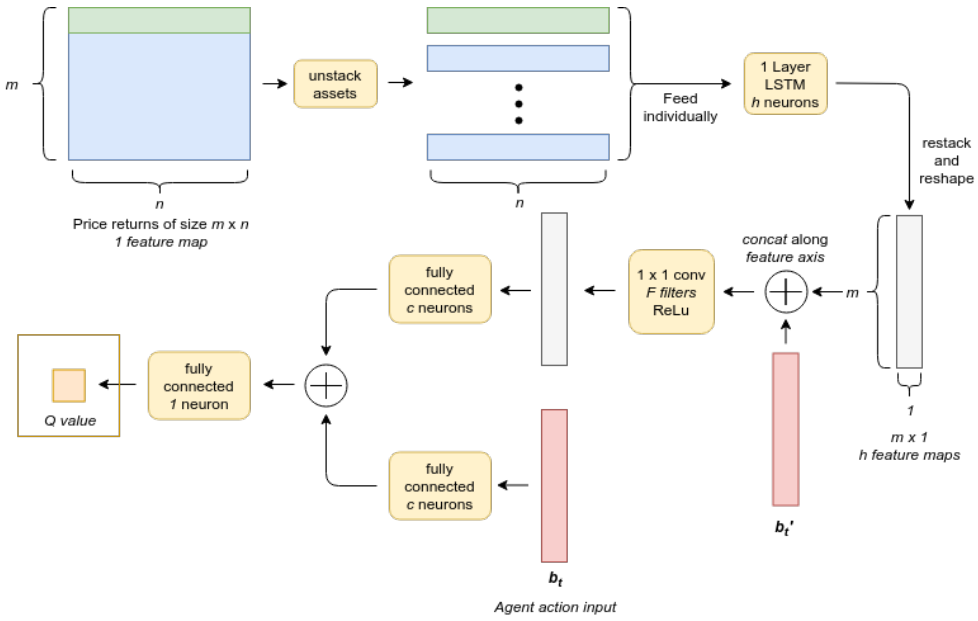

Figure 18: RNN critic network using the EIIE paradigm. This network uses the RNN actor network as its base. Similarly to the CNN critic, we do not restrict the 1×1 convolution to one feature map, and combine state and action information using fully connected networks.

.

algorithm using CNN-based actor and critic architectures with various history lengths. We use $\gamma = 0.5$, $n = 2$, no auxiliary learning tasks, and train our agents for $200,000$ iterations each. The training set performance, as seen in Figure 20 indicates that all models are able to fit to the training set, with a history length of seven performing the best. Figure 21, however, indicates that history lengths above nine overfit, while a history length of size three performs optimally. Agents using a history length of three, seven, and nine significantly outpace the market value.

Analogous results are shown for LSTM networks in Figures 22 and 23. We see similar patterns as those demonstrated with CNN-based agents – higher history lengths tend to overfit, while window lengths of $3, 7$ and 9 are able to generalize to the test set. LSTM agents also tend to fit the training data better, as seen in Figure 22.

We also evaluate agents trained with various values of γ. Training and testing performance are shown for CNN agents in Figures 24 and 25, respectively. We use a

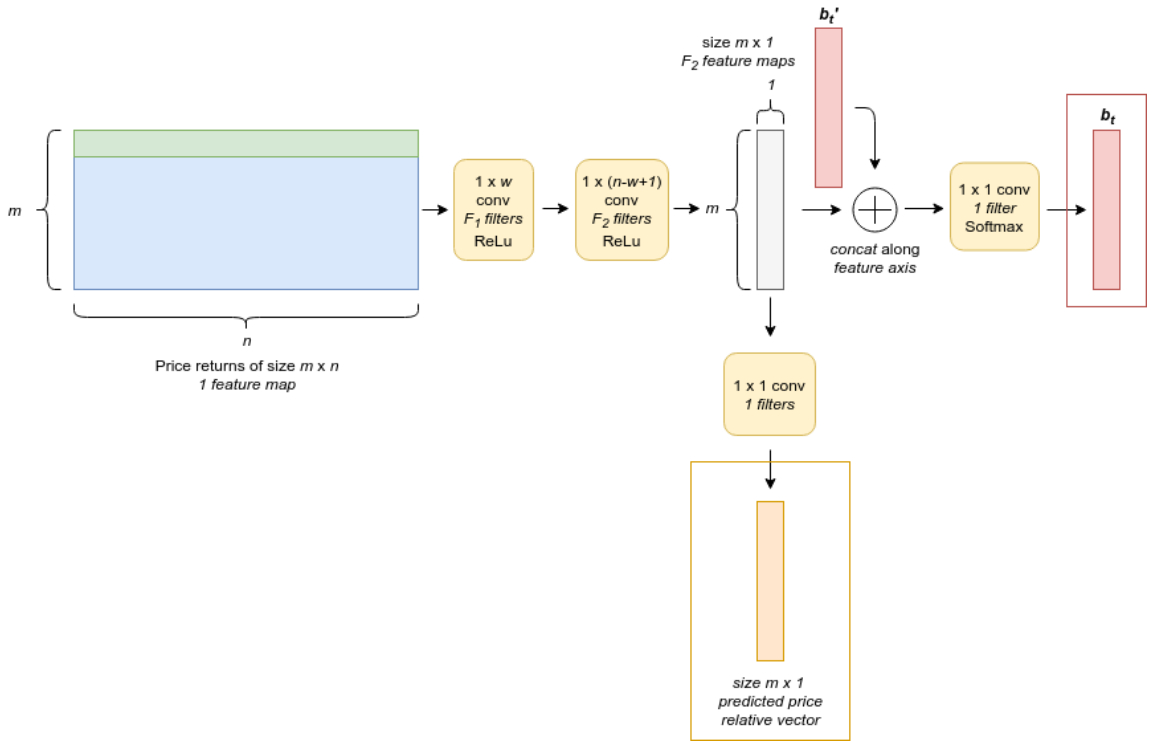

Figure 19: CNN actor network with auxiliary losses. A 1×1 convolutional layer with 1 filter predicts the price relative vector $\mathbf{x}_{t+1}$ of the subsequent time period using solely features derived from the price returns fed to the actor network. We do not incorporate $\mathbf{b}_t^{'}$ into this prediction because $\mathbf{x}_{t+1}$ has no dependence with $\mathbf{b}_t^{'}$. Note that this convolutional layer uses a linear activation function..

.

history length of 3, $n = 2$, no auxiliary learning tasks, and train agents for $200,000$ iterations each. Note how while γ values of $0.1, 0.25$ and 0.5 are able to perform well on the training set, a γ value of 0.99 ends up performing poorly. It was observed during training, however, that performance for $\gamma = 0.99$ *degraded* after hitting a maximal performance around $40,000$ iterations. This can be attributed to the instability caused by the excessive bootstrapping of $\gamma = 0.99$. We can observe that $\gamma = 0.25$ performs best on the test set, compared to the $\gamma = 0$ approach taken by [10]. This is in alignment with the recognition that the reward signal is recurrent in nature and that future actions are decreasingly dependent on previous actions.

Similar results are shown for LSTM-based agents with a history length of 3 in

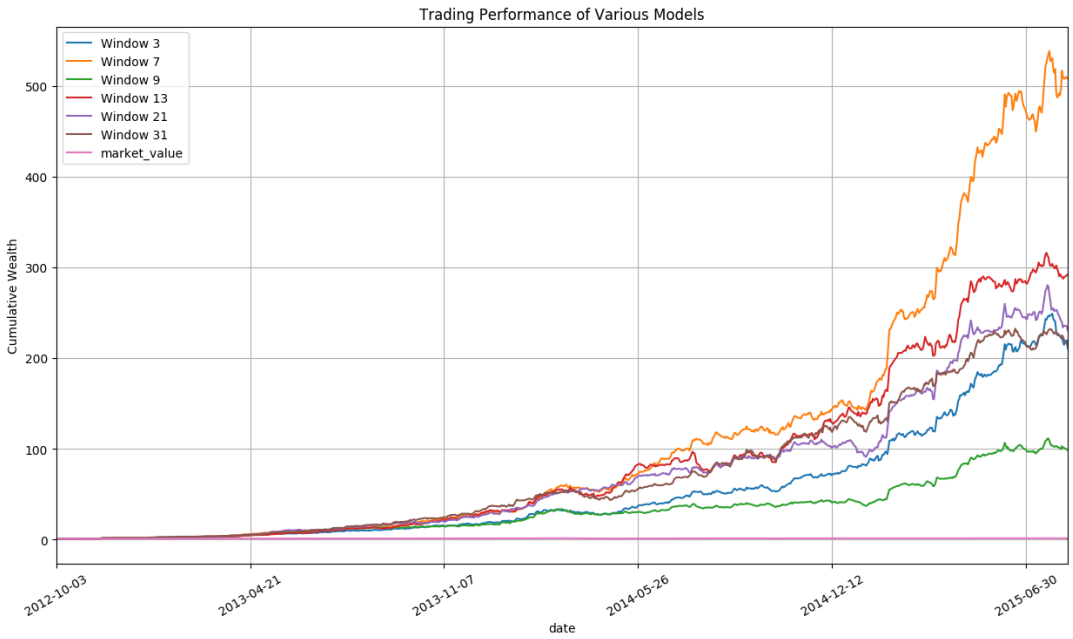

Figure 20: Training set performance of various CNN-based agents with different history lengths.

.

Figures 26 and 27. LSTM-based agents do not suffer the same boostrapping problems associated with high γ values that afflict CNN-based agents, as their structure allows them to retain information for longer time steps.

We also vary n to test the impact of variance on learned policies. We use a history length of 3, $\gamma = 0.25$, and no auxiliary learning tasks for all agents. Training and testing performance of CNN-based agents are shown in Figures 28 and 29. We see that while $n = 1$ and $n = 2$ match each other in performance on the training set, $n = 4$ performs substantially worse. These results are mirrored on the testing set. We believe that the additional variance caused increasing values of n outweighs the stability measures imposed by using a replay buffer and soft target networks.

We show similar results for LSTM-based agents, where we vary n for $\gamma = 0.25$ and a window length of 3. No auxiliary learning tasks are used for these agents. LSTM agents are observed to be more robust to the additional variance introduced by higher n values.

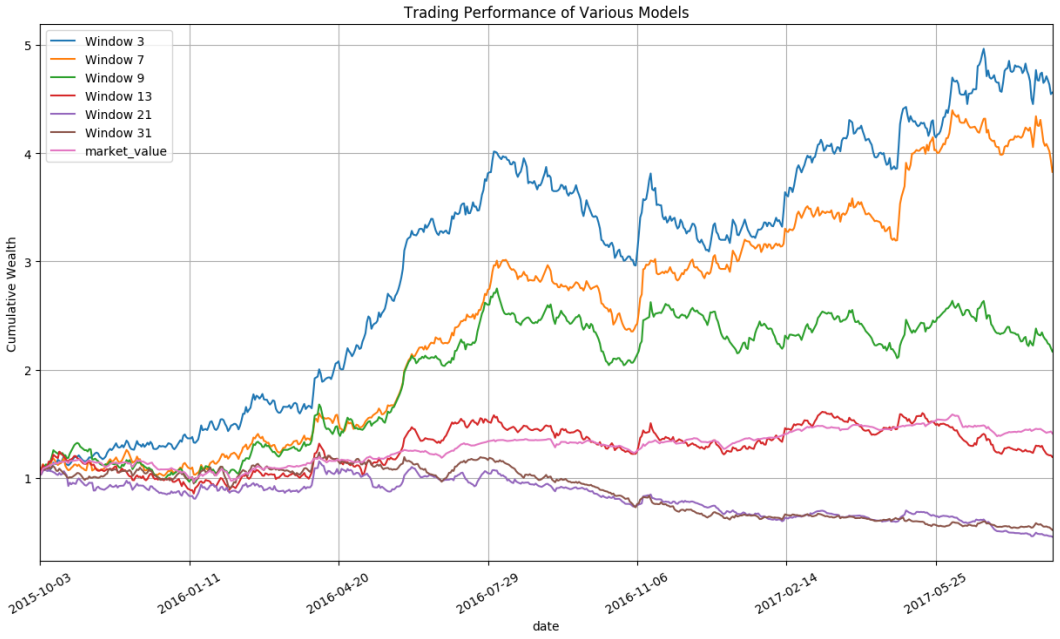

Figure 21: Testing set performance of various CNN-based agents with different history lengths.

.

Lastly, we experiment with different strengths for our auxiliary tasks. We show training and testing results for agents using $n = 1$, $\gamma = 0.25$ and a window length of 3 in Figures 32 and 33. We can observe from Figures 32 through 35 that auxiliary learning tasks *can* bootstrap learning, depending on where they are applied. The auxiliary task of minimizing commission significantly hampers learning and generalization performance, while the auxiliary task of predicting the next price relative vector is able to successfully bootstrap the training process for a number of different strengths. As expected, however, the generalization performance of models trained with auxiliary tasks is largely sub-optimal, due to early convergence.

Finally, we compare our agents' performance versus that of standard benchmark algorithms. We compare versus the *Best Stock*, *Anti Correlation*, *Online Moving Average Reversion*, and *Online Newton Step* algorithms defined in Li and Hoi 2014 ([14]). We use Jiang et al. 2017's implementation of these standard algorithms. These comparisons are shown in Figure 36 and it can be seen that our agent outperforms the

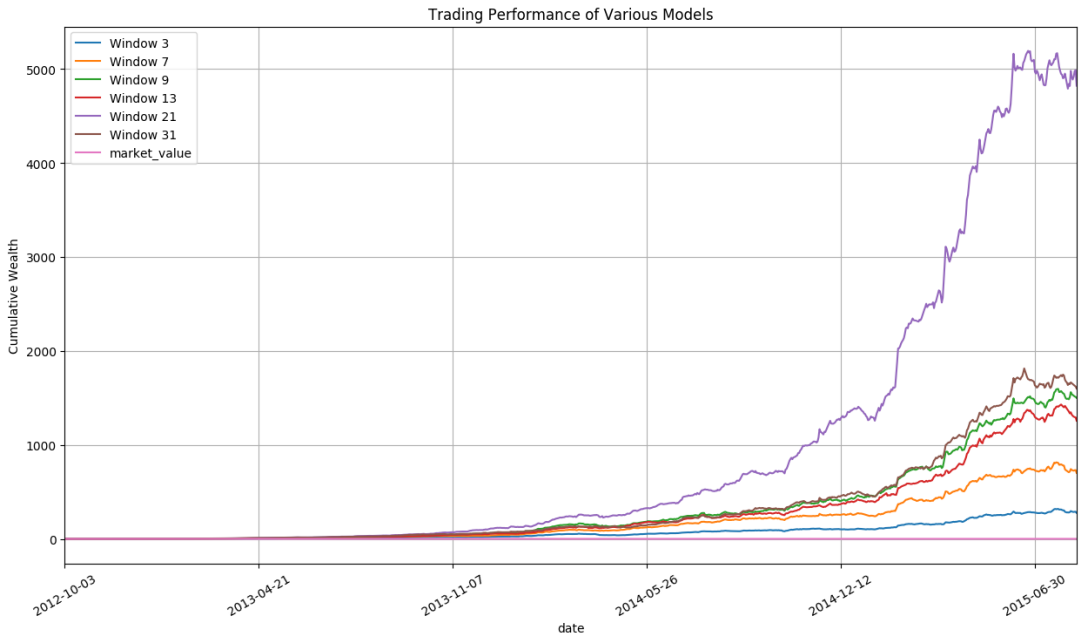

Figure 22: Training set performance of various LSTM-based agents with different history lengths.

.

standard algorithms by a significant amount for a variety of auxiliary task strengths. Although our agent does not beat these standard models by a similar amount as Jiang et al. 2017, it is tough to compare the results since their agents trade on the cryptocurrency markets, an inherently more volatile market than our S&P 500 dataset. Rather, these results indicate that a more standard deep-RL based approach using state-of-the-art deep continuous control methods can be applied to the portfolio management problem. We tally all results in Tables 1 through 5, and report the final accumulated portfolio value (fAPV), maximum drawdown (MDD), and Sharpe ratio for each of these agents. We also report these same metrics for standard benchmark algorithms.

$\gamma = 0.5, n = 2$, w variable

Model type	**w**	**fAPV**	**MDD**	**Sharpe**
CNN	3	4.56	-0.117	0.774
CNN	7	3.83	-0.119	0.709
CNN	9	2.17	-0.111	0.416
CNN	13	1.19	-0.119	0.131
CNN	21	0.457	-0.114	-0.326
CNN	31	0.520	-0.117	-0.304
LSTM	3	4.18	-0.116	0.699
LSTM	7	4.48	-0.115	0.752
LSTM	9	3.39	-0.088	0.727
LSTM	13	1.31	-0.110	0.172
LSTM	21	1.61	-0.114	-0.283
LSTM	31	2.39	-0.109	0.454

Table 1: CNN and LSTM agents with variable history lengths

$w = 3, n = 2$, γ variable

Model type	γ	**fAPV**	**MDD**	**Sharpe**
CNN	0.1	5.45	-0.156	0.778
CNN	0.25	7.26	-0.120	0.963
CNN	0.5	4.56	-0.116	0.774
CNN	0.99	0.437	-0.095	-0.387
LSTM	0.1	5.29	-0.111	0.842
LSTM	0.25	3.25	-0.111	0.596
LSTM	0.5	4.18	-0.116	0.699
LSTM	0.99	4.12	-0.110	0.700

Table 2: CNN and LSTM agents with variable γ

$w = 3, \gamma = 0.25, n$ variable

Model type	**n**	**fAPV**	**MDD**	**Sharpe**
CNN	1	9.15	-0.113	1.02
CNN	2	7.26	-0.120	0.963
CNN	4	2.34	-0.121	0.412
LSTM	1	6.53	-0.113	0.931
LSTM	2	3.24	-0.111	0.596
LSTM	4	5.91	-0.110	0.870

Table 3: CNN and LSTM agents with variable n

$w = 3, n = 1, \gamma = 0.25$, with auxiliary tasks

Model type	auxiliary strengths	**fAPV**	**MDD**	**Sharpe**
CNN	(0, 0, 0)	9.15	-0.113	1.02
CNN	(0, 0, 0.1)	3.38	-0.152	0.592
CNN	(0, 0, 1.0)	2.63	-0.152	0.464
CNN	(0, 0.1, 0.0)	2.63	-0.152	0.502
CNN	(0, 0.1, 0.1)	5.34	-0.156	0.798
CNN	(0, 1, 0)	7.40	-0.151	0.944
CNN	(1, 0, 0)	1.01	-0.002	0.444
CNN	(0, 0, 10)	8.19	-0.117	0.983
CNN	(0, 10, 0)	4.67	-0.118	0.773
CNN	(0, 1, 1)	7.62	-0.156	0.934
LSTM	(0, 0, 0)	6.53	-0.113	0.932
LSTM	(0, 0, 0.1)	4.31	-0.119	0.716
LSTM	(0, 0, 1.0)	4.70	-0.114	0.741
LSTM	(0, 0.1, 0.0)	5.50	-0.116	0.865
LSTM	(0, 0.1, 0.1)	5.73	-0.116	0.861
LSTM	(0, 1, 0)	6.47	-0.116	0.942
LSTM	(1, 0, 0)	1.01	-0.002	0.458
LSTM	(0, 0, 10)	5.04	-0.111	0.780
LSTM	(0, 10, 0)	4.03	-0.112	0.704
LSTM	(0, 1, 1)	7.21	-0.113	0.942

Table 4: CNN and LSTM agents with variable auxiliary task strengths. The strengths of the tasks are arranged in order in the second column, according to the ordering of Paragraph 4.0.0.3.

Benchmark algorithms

Model type	**fAPV**	**MDD**	**Sharpe**
Best Stock	1.94	-0.194	0.372
Anticor	1.16	-0.090	0.157
OLMAR	0.31	-0.236	-0.501
ONS	1.52	-0.061	0.407

Table 5: Results for standard benchmark algorithms.

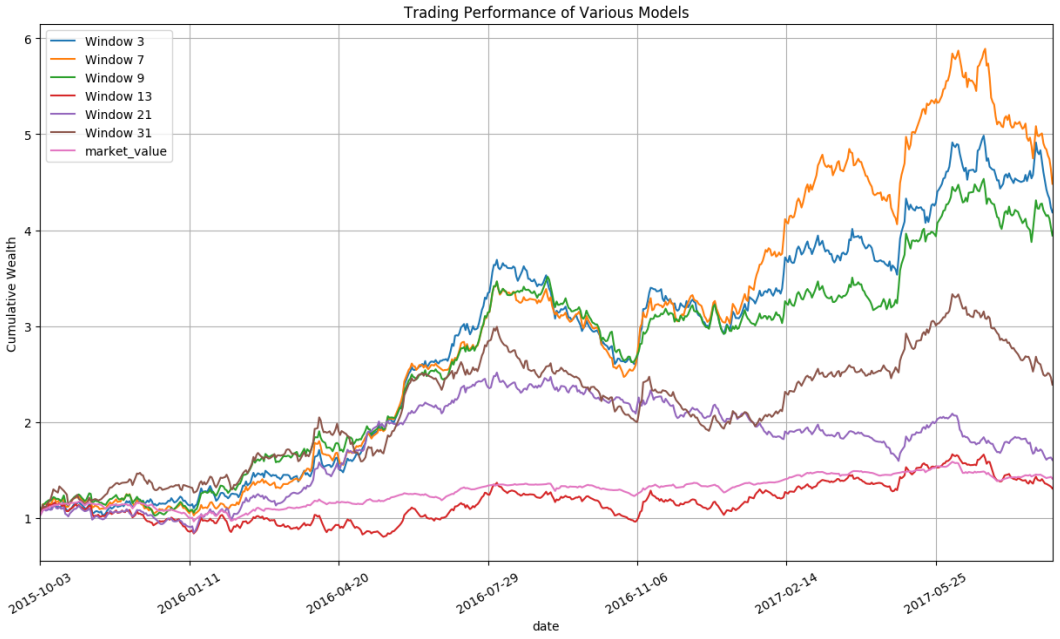

Figure 23: Testing set performance of various LSTM-based agents with different history lengths.

.

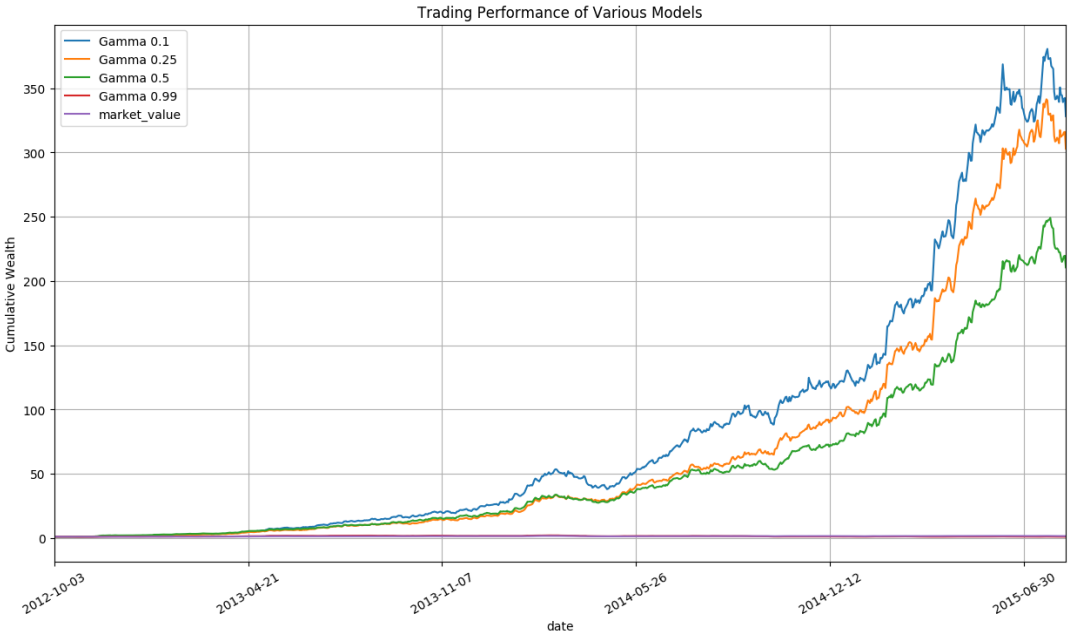

Figure 24: Training set performance of CNN-based agents for various values of γ.

.

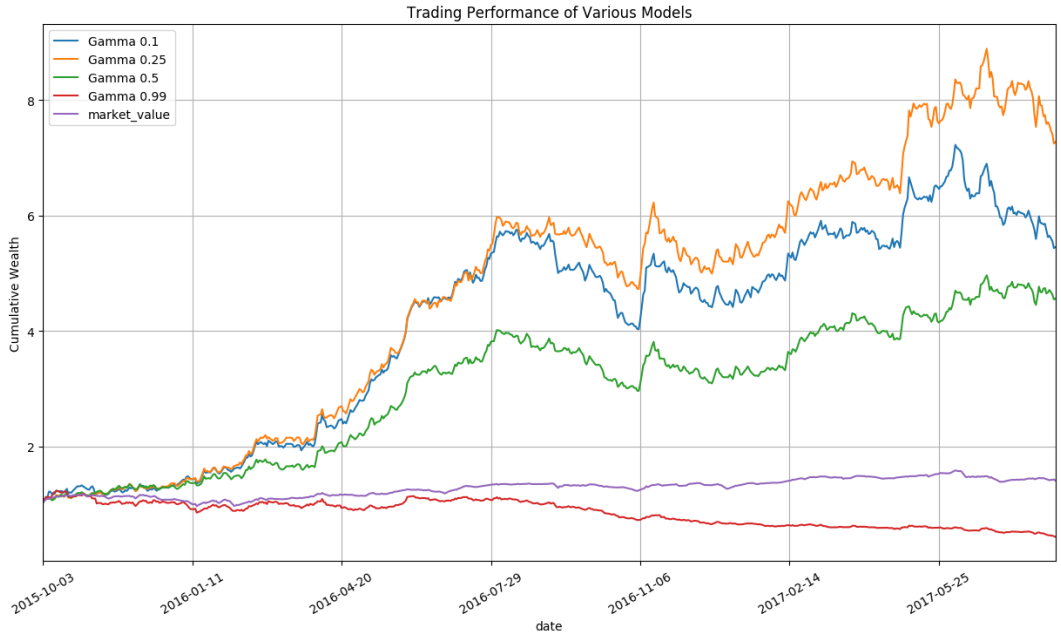

Figure 25: Test set performance of CNN-based agents for various values of γ.

.

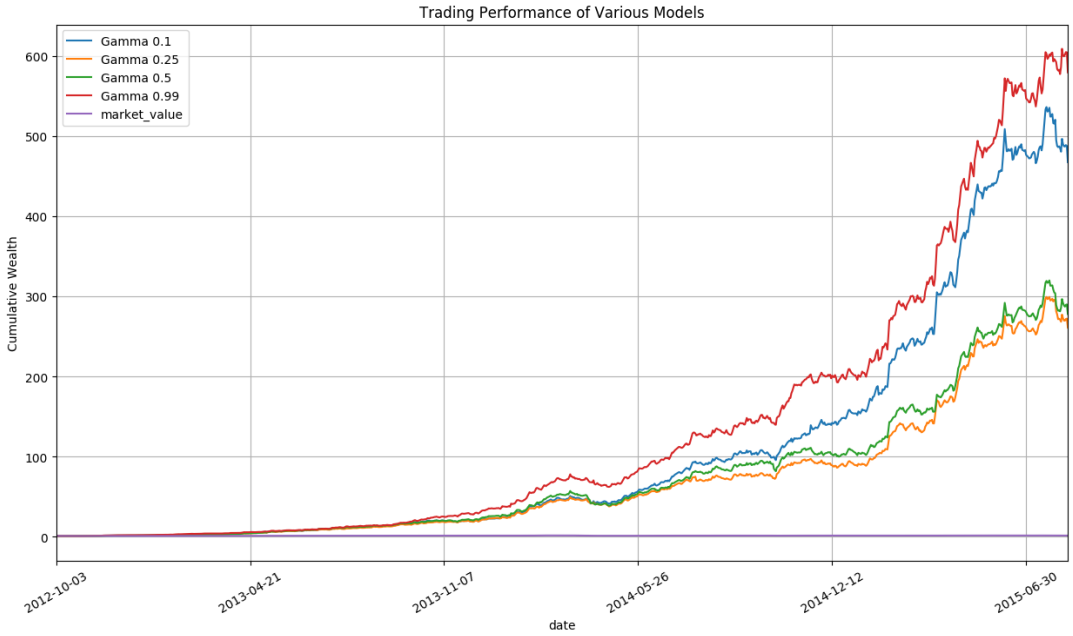

Figure 26: Training set performance of various LSTM-based agents with different γ's.

.

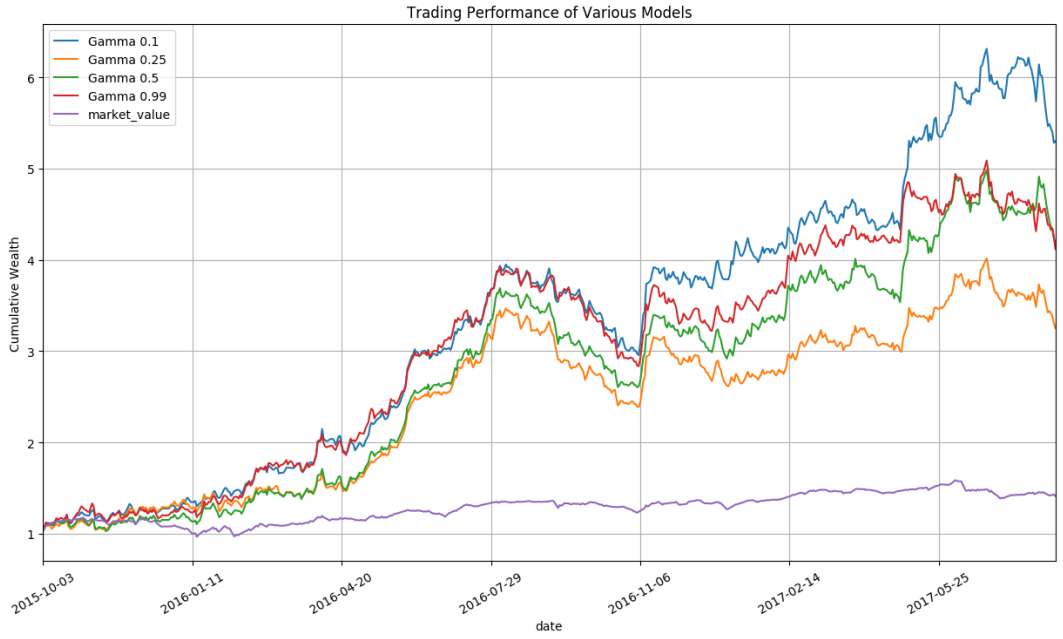

Figure 27: Testing set performance of various LSTM-based agents with different γ's.

.

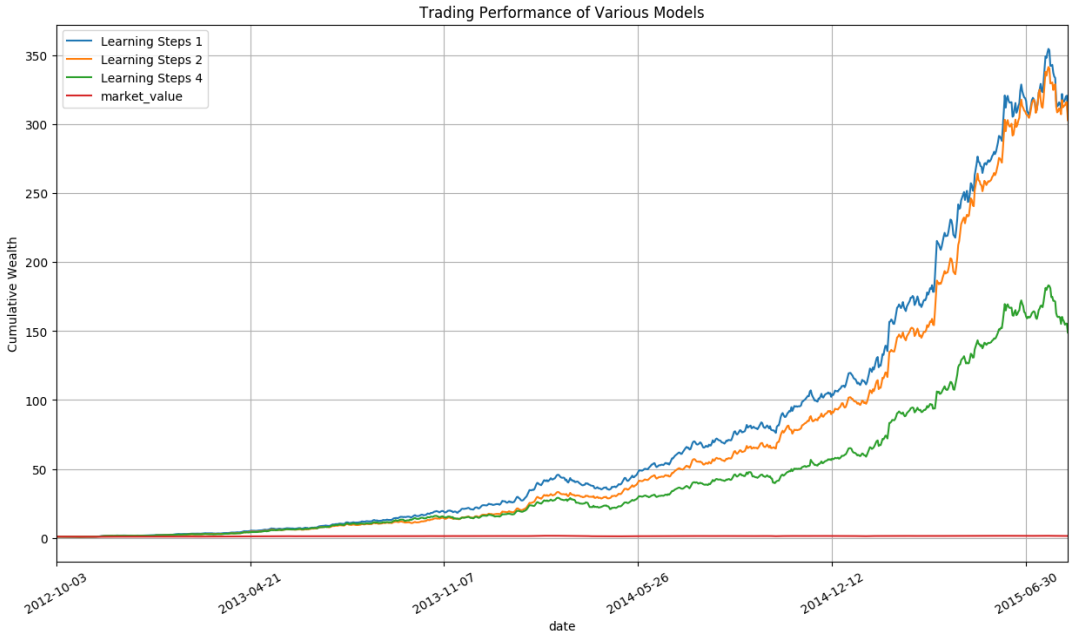

Figure 28: Training set performance of CNN-based agents for various values of n.

.

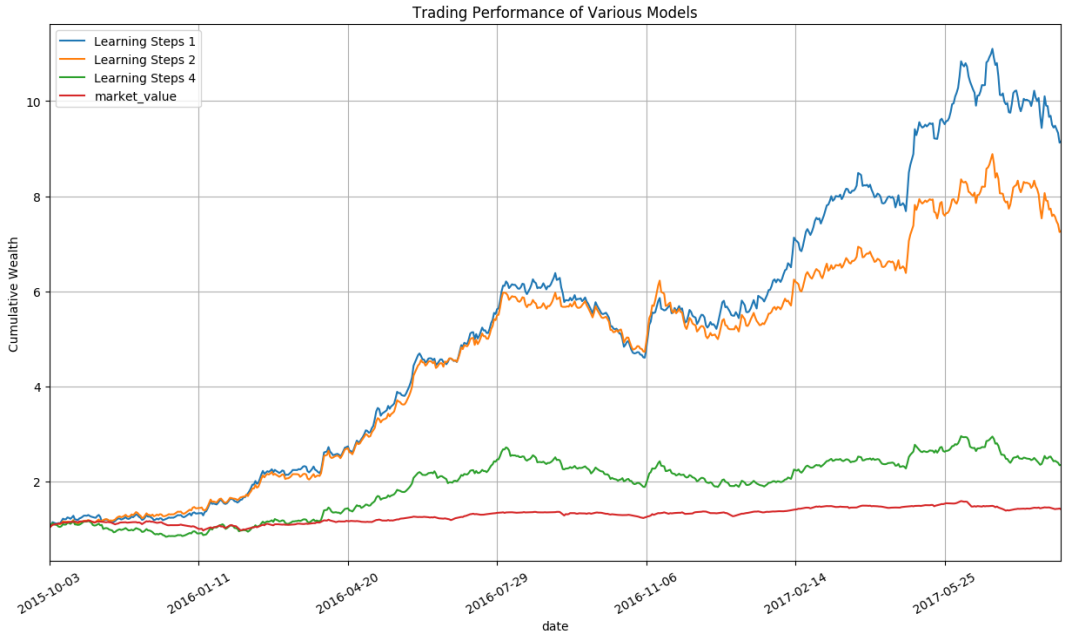

Figure 29: Test set performance of CNN-based agents for various values of n.

.

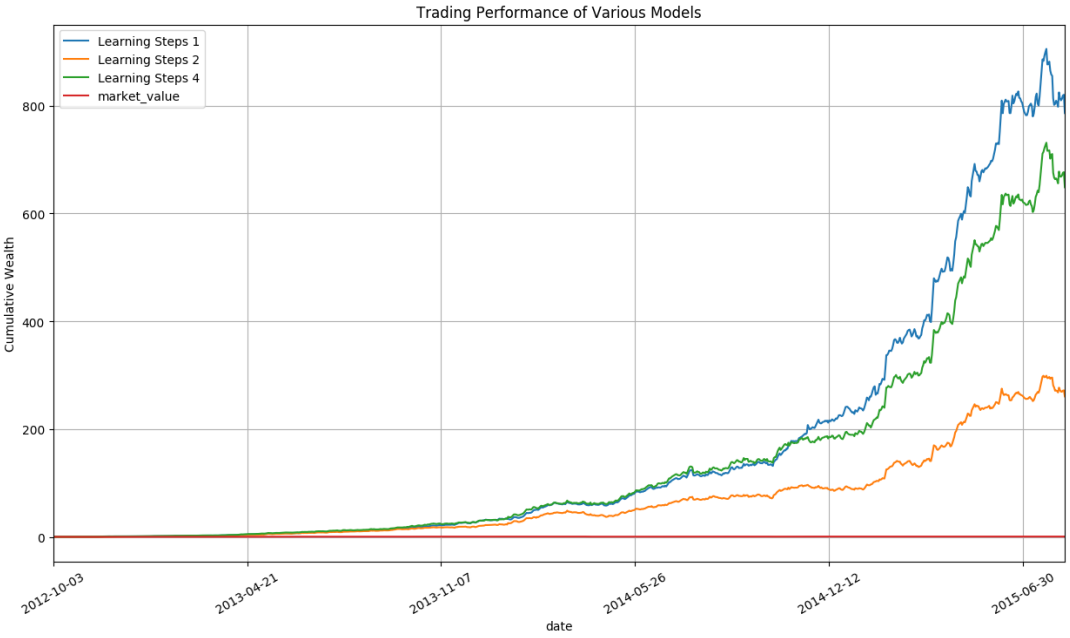

Figure 30: Training set performance of LSTM-based agents for various values of n.

.

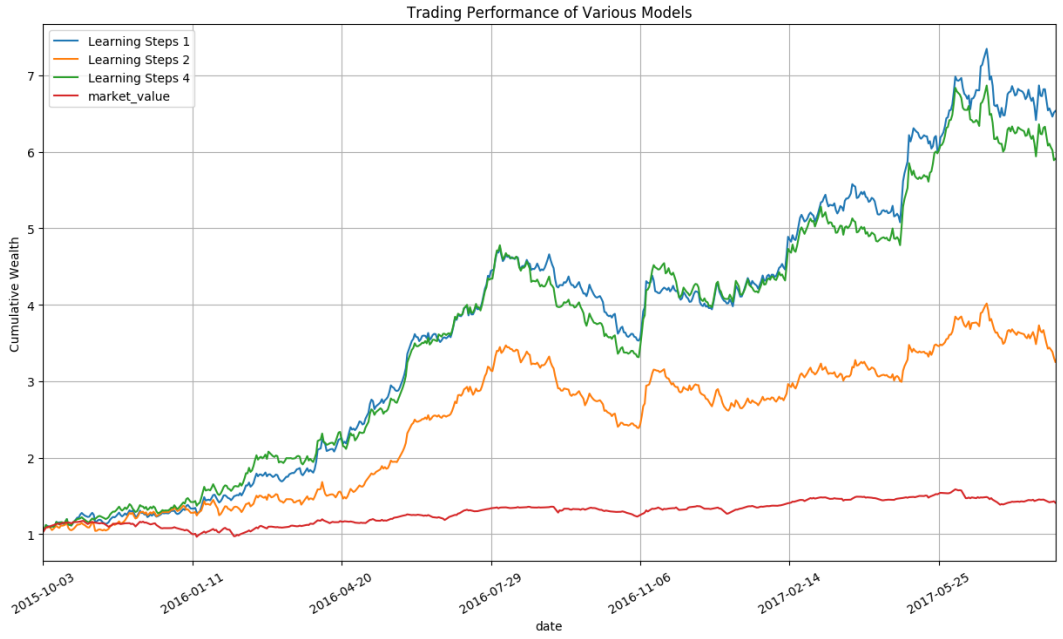

Figure 31: Test set performance of LSTM-based agents for various values of n.

.

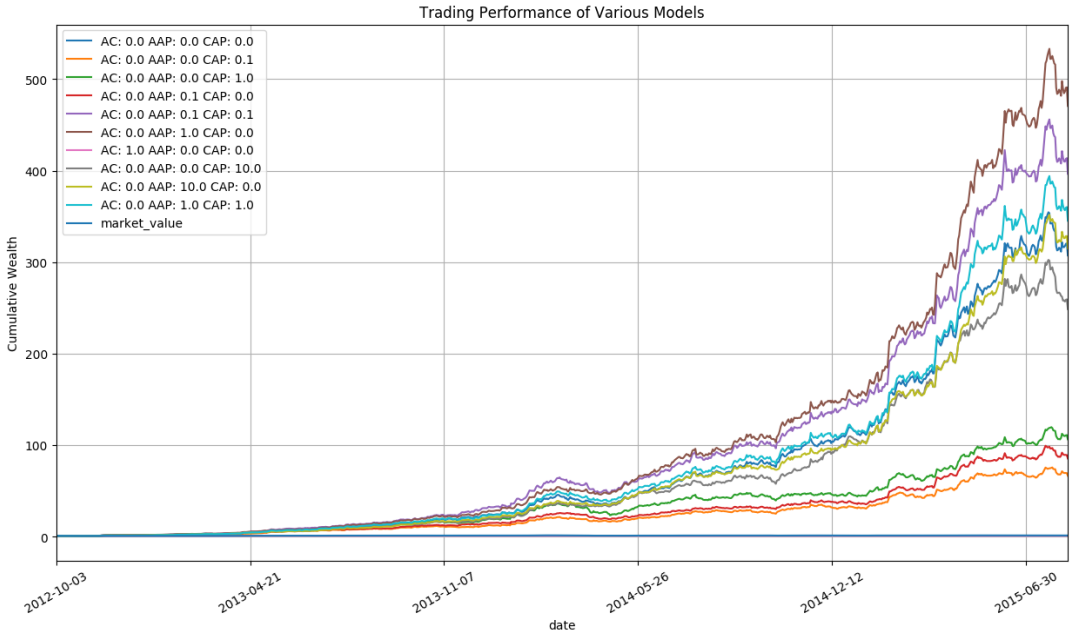

Figure 32: Training set performance of CNN-based agents for various auxiliary learning task strengths.

.

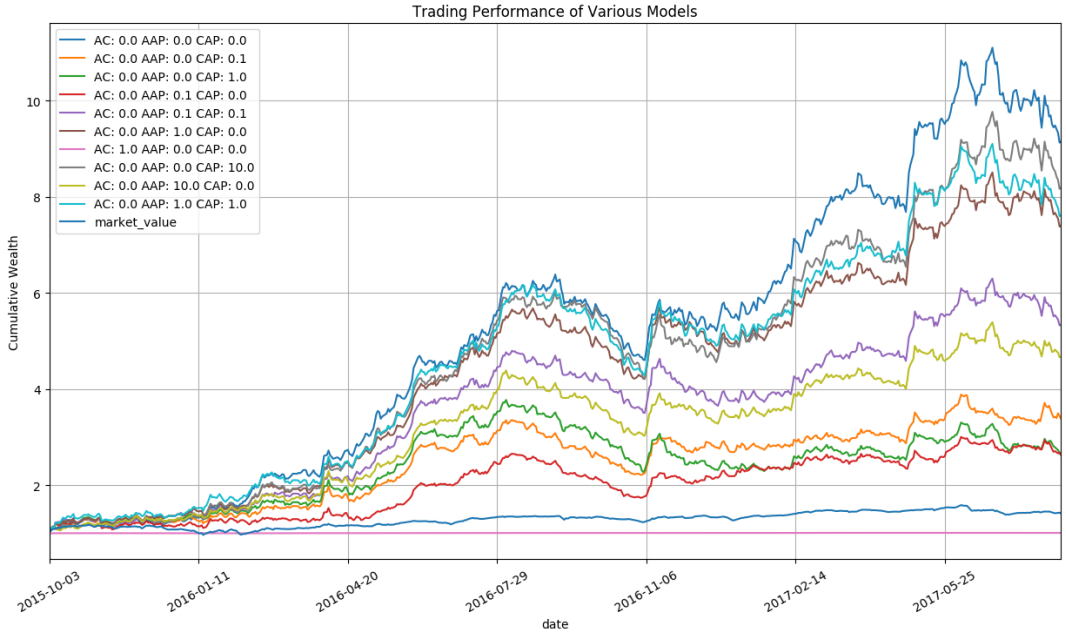

Figure 33: Test set performance of CNN-based agents for various values auxiliary learning task strengths.

.

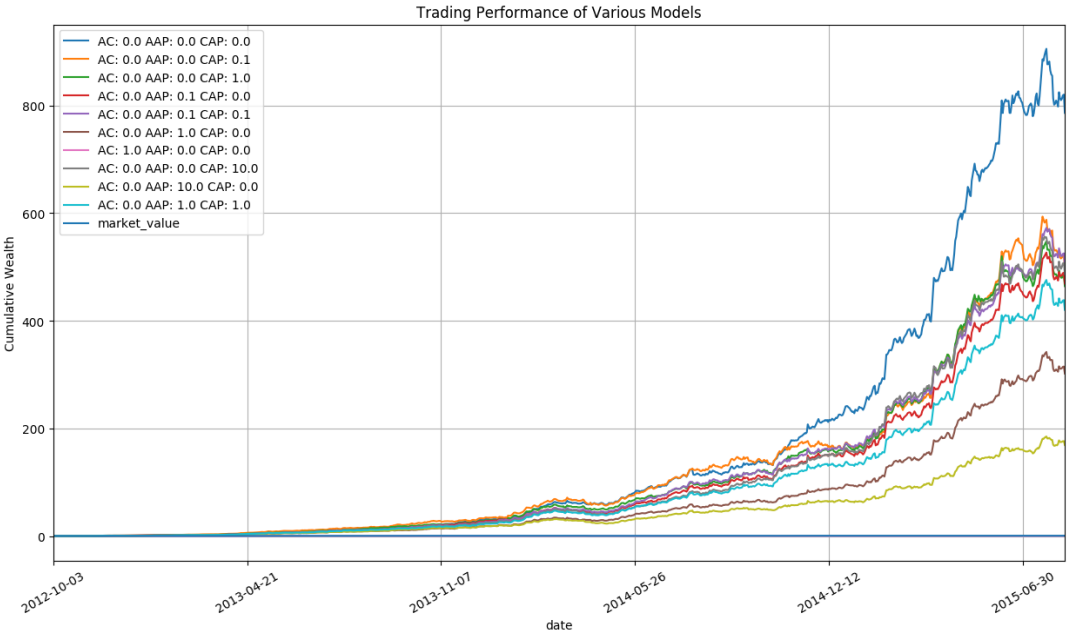

Figure 34: Training set performance of LSTM-based agents for various auxiliary learning task strengths.

.

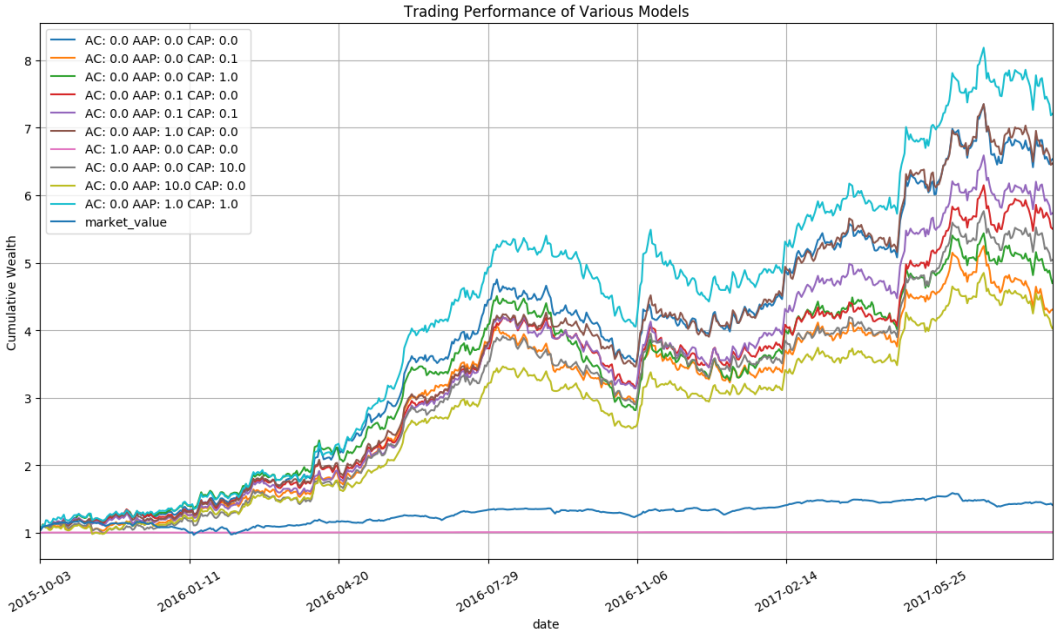

Figure 35: Test set performance of LSTM-based agents for various auxiliary learning task strengths.

.

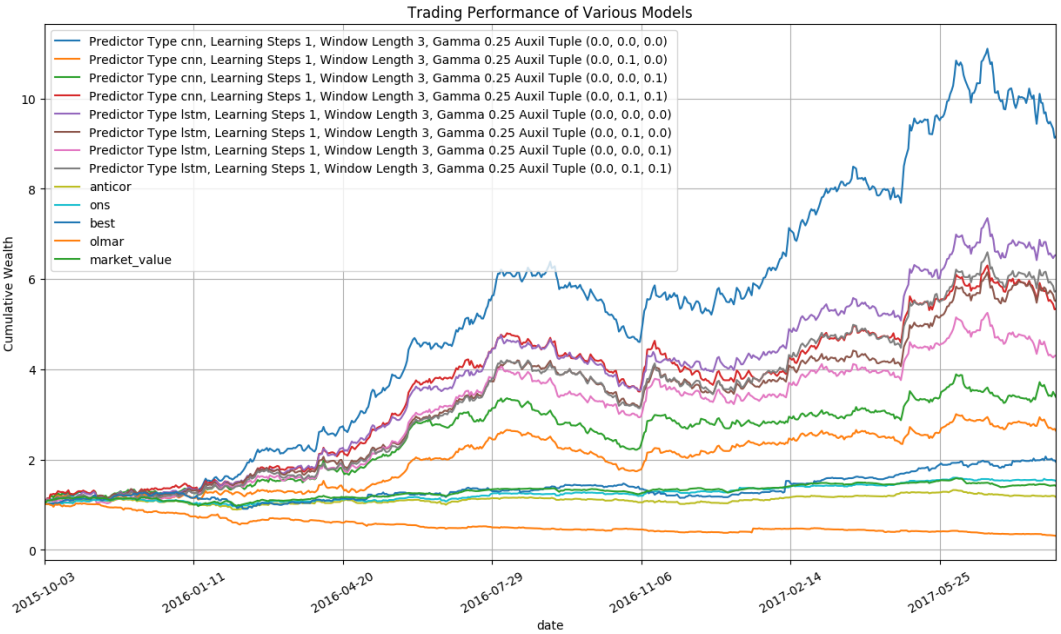

Figure 36: Test set performance of LSTM-based agents for various auxiliary learning task strengths.

.

5 Conclusion & Future Work

Reinforcement learning (RL) continues to be applied successfully to many difficult problem scenarios due to innovations in both RL and deep learning. We have presented a technique that utilizes a state-of-the-art deep continuous control method, Deep Deterministic Policy Gradient (DDPG) [16], to perform the portfolio management task. Furthermore, we have augmented DDPG with both $n - step$ rollouts and with auxiliary learning tasks. While $n - step$ rollouts were unable to provide performance gains due to the inherent volatility of the environment, auxiliary learning tasks were able to bootstrap training, with the side effect of hampering generalization, as predicted. Our agent is able to significantly outperform standard benchmark online portfolio management algorithms.

Much more, however, can still be accomplished to expand the results of this work. Concerning DDPG itself, one can leverage the results of Double DQN in tandem with Dueling Networks, as discussed in section 1.4.2 to obtain more accurate critic estimates. Furthermore, a large portion of the actor and critic parameters can be shared, allowing for greater generalization capabilities, specifically, all parameters computing the growth assessment vectors of both the actor and critic architectures. Prioritized Experience Replay [29] can also be leveraged to improve training speed, while Generalized Advantage Estimation [30] can be used as a more general form of our $n - step$ algorithm. Although we have used DDPG in this thesis, Trust Region Policy Optimization (TRPO) [31] provides an exciting avenue of future research due to its performance in unstable environments. Lastly, we can also explore the usage of various reward signals, such as the differential Sharpe ratio, so that our agent can learn to maximize other criteria rather than just pure profit.

References

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN: 0387310738.

[2] James Cumming. "An Investigation into the Use of Reinforcement Learning Techniques within the Algorithmic Trading Domain". In: (June 2015). URL: https://www.doc.ic.ac.uk/teaching/distinguished-projects/2015/j.cumming.pdf.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[4] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *CoRR* abs/1509.06461 (2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

[5] Matthew J. Hausknecht and Peter Stone. "Deep Recurrent Q-Learning for Partially Observable MDPs". In: *CoRR* abs/1507.06527 (2015). arXiv: 1507.06527. URL: http://arxiv.org/abs/1507.06527.

[6] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *CoRR* abs/1502.01852 (2015). arXiv: 1502.01852. URL: http://arxiv.org/abs/1502.01852.

[7] Kaiming He et al. "Mask R-CNN". In: *CoRR* abs/1703.06870 (2017). arXiv: 1703.06870. URL: http://arxiv.org/abs/1703.06870.

[8] Peter Henderson et al. "Deep Reinforcement Learning that Matters". In: *CoRR* abs/1709.06560 (2017). arXiv: 1709.06560. URL: http://arxiv.org/abs/1709.06560.

[9] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[10] Zhengyao Jiang, Dixing Xu, and Jinjun Liang. “A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem”. In: *CoRR* abs/1706.10059 (2017). arXiv: 1706.10059. URL: http://arxiv.org/abs/1706.10059.

[11] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *CoRR* abs/1412.6980 (2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

[12] Anders Krogh and John A. Hertz. “A Simple Weight Decay Can Improve Generalization”. In: *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 4*. Morgan Kaufmann, 1992, pp. 950–957.

[13] Yann LeCun, Y Bengio, and Geoffrey Hinton. “Deep Learning”. In: 521 (May 2015), pp. 436–44.

[14] Bin Li and Steven C. H. Hoi. “Online Portfolio Selection: A Survey”. In: *ACM Comput. Surv.* 46.3 (Jan. 2014), 35:1–35:36. ISSN: 0360-0300. DOI: 10.1145/2512962. URL: http://doi.acm.org/10.1145/2512962.

[15] Yuxi Li. “Deep Reinforcement Learning: An Overview”. In: *CoRR* abs/1701.07274 (2017). arXiv: 1701.07274. URL: http://arxiv.org/abs/1701.07274.

[16] Timothy P. Lillicrap et al. “Continuous control with deep reinforcement learning”. In: *CoRR* abs/1509.02971 (2015). arXiv: 1509.02971. URL: http://arxiv.org/abs/1509.02971.

[17] Long-Ji Lin. “Self-improving reactive agents based on reinforcement learning, planning and teaching”. In: *Machine Learning* 8.3 (May 1992), pp. 293–321. ISSN: 1573-0565. DOI: 10.1007/BF00992699. URL: https://doi.org/10.1007/BF00992699.

[18] Piotr Mirowski et al. “Learning to Navigate in Complex Environments”. In: *CoRR* abs/1611.03673 (2016). arXiv: 1611.03673. URL: http://arxiv.org/abs/1611.03673.

[19] Volodymyr Mnih et al. “Asynchronous Methods for Deep Reinforcement Learning”. In: *CoRR* abs/1602.01783 (2016). arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783.

[20] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[21] Volodymyr Mnih et al. “Playing Atari With Deep Reinforcement Learning”. In: *NIPS Deep Learning Workshop*. 2013.

[22] John Moody et al. “Performance Functions and Reinforcement Learning for Trading Systems and Portfolios”. In: 1998.

[23] Vinod Nair and Geoffrey E. Hinton. “Rectified Linear Units Improve Restricted Boltzmann Machines”. In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML’10. Haifa, Israel: Omnipress, 2010, pp. 807–814. ISBN: 978-1-60558-907-7. URL: http://dl.acm.org/citation.cfm?id=3104322.3104425.

[24] Pierpaolo Necchi. “Policy Gradient Algorithms for the Asset Allocation Problem”. In: (2016). URL: https://github.com/pnecchi/Thesis/blob/master/MS_Thesis_Pierpaolo_Necchi.pdf.

[25] Andrew Y. Ng. “Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance”. In: *Proceedings of the Twenty-first International Conference on Machine Learning*. ICML ’04. Banff, Alberta, Canada: ACM, 2004, pp. 78–. ISBN: 1-58113-838-5. DOI: 10.1145/1015330.1015435. URL: http://doi.acm.org/10.1145/1015330.1015435.

[26] Christopher Olah. *Understanding LSTM Networks*. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[27] Herbert Robbins and Sutton Monro. "A Stochastic Approximation Method". In: *Ann. Math. Statist.* 22.3 (Sept. 1951), pp. 400–407. DOI: 10.1214/aoms/1177729586. URL: https://doi.org/10.1214/aoms/1177729586.

[28] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1". In: ed. by David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group. Cambridge, MA, USA: MIT Press, 1986. Chap. Learning Internal Representations by Error Propagation, pp. 318–362. ISBN: 0-262-68053-X. URL: http://dl.acm.org/citation.cfm?id=104279.104293.

[29] Tom Schaul et al. "Prioritized Experience Replay". In: *CoRR* abs/1511.05952 (2015). arXiv: 1511.05952. URL: http://arxiv.org/abs/1511.05952.

[30] John Schulman et al. "High-Dimensional Continuous Control Using Generalized Advantage Estimation". In: *CoRR* abs/1506.02438 (2015). arXiv: 1506.02438. URL: http://arxiv.org/abs/1506.02438.

[31] John Schulman et al. "Trust Region Policy Optimization". In: *CoRR* abs/1502.05477 (2015). arXiv: 1502.05477. URL: http://arxiv.org/abs/1502.05477.

[32] David Silver. "Introduction to Reinforcement Learning". University Lecture. 2014.

[33] David Silver. "Markov Decision Processes". University Lecture. 2014.

[34] David Silver. "Model-Free Control". University Lecture. 2014.

[35] David Silver. "Planning by Dynamic Programming". University Lecture. 2014.

[36] David Silver. "Value Function Approximation". University Lecture. 2014.

[37] David Silver et al. “Deterministic Policy Gradient Algorithms.” In: *ICML*. Vol. 32. JMLR Workshop and Conference Proceedings. JMLR.org, 2014, pp. 387–395. URL: http://dblp.uni-trier.de/db/conf/icml/icml2014.html#SilverLHDWR14.

[38] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[39] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning : An Introduction*. MIT Press, 1998.

[40] Ziyu Wang, Nando de Freitas, and Marc Lanctot. “Dueling Network Architectures for Deep Reinforcement Learning”. In: *CoRR* abs/1511.06581 (2015). arXiv: 1511.06581. URL: http://arxiv.org/abs/1511.06581.

[41] Ronald J. Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine Learning* 8.3 (May 1992), pp. 229–256. ISSN: 1573-0565. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696.

[42] Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. “On Early Stopping in Gradient Descent Learning”. In: *Constructive Approximation* 26.2 (Aug. 2007), pp. 289–315. ISSN: 1432-0940. DOI: 10.1007/s00365-006-0663-2. URL: https://doi.org/10.1007/s00365-006-0663-2.

[43] Chi Zhang and Corey Chen. *Deep Reinforcement Learning for Portfolio Management*. URL: http://www-scf.usc.edu/~zhan527/post/cs599/.

6 Appendix - Selected Code

stock_trading.py

```
"""
Modified from https://github.com/vermouth1992/drl-portfolio-management/blob/master/src/stock_trading.py
"""

from __future__ import print_function, division

from model.ddpg.actor import ActorNetwork
from model.ddpg.critic import CriticNetwork
from model.ddpg.ddpg import DDPG
from model.ddpg.ornstein_uhlenbeck import OrnsteinUhlenbeckActionNoise

from environment.portfolio import PortfolioEnv
from utils.data import read_stock_history, read_stock_history_csvs, normalize

import argparse
import numpy as np
import tflearn
import tensorflow as tf
import pandas as pd
import pprint
import utils.datacontainer

DEBUG = True


def get_model_path(window_length, predictor_type, use_batch_norm, learning_steps=0, gamma=0.5,
                   auxiliary_commission=0, actor_auxiliary_prediction=0, critic_auxiliary_prediction=0):
    if use_batch_norm:
        batch_norm_str = 'batch_norm'
    else:
        batch_norm_str = 'no_batch_norm'

    learning_steps_str = 'learning_steps_'+str(learning_steps)
    gamma_str = 'gamma_'+str(gamma)
    auxiliary_str = 'ac_{}_aap_{}_cap_{}'.format(str(float(auxiliary_commission)),
                                                 str(float(actor_auxiliary_prediction)),
                                                 str(float(critic_auxiliary_prediction)))

    return 'weights/{}/window_{}/{}/{}/{}/{}/'.format(predictor_type, window_length, batch_norm_str,
                                                      learning_steps_str, gamma_str, auxiliary_str)


def get_result_path(window_length, predictor_type, use_batch_norm, learning_steps=0, gamma=0.5,
                    auxiliary_commission=0, actor_auxiliary_prediction=0, critic_auxiliary_prediction=0):
    if use_batch_norm:
        batch_norm_str = 'batch_norm'
    else:
        batch_norm_str = 'no_batch_norm'

    learning_steps_str = 'learning_steps_'+str(learning_steps)
    gamma_str = 'gamma_'+str(gamma)
    auxiliary_str = 'ac_{}_aap_{}_cap_{}'.format(str(float(auxiliary_commission)),
                                                 str(float(actor_auxiliary_prediction)),
                                                 str(float(critic_auxiliary_pred
```

```
    iction)))
55
56      return 'results/{}/window_{}/{}/{}/{}/{}/'.format(predictor_type, window_length, batc
    h_norm_str,
57                                                        learning_steps_str, gamma_
    str, auxiliary_str)
58
59  def get_infer_path(window_length, predictor_type, use_batch_norm, learning_steps
    =0, gamma=0.5,
60                     auxiliary_commission=0, actor_auxiliary_prediction=0, critic
    _auxiliary_prediction=0):
61      if use_batch_norm:
62          batch_norm_str = 'batch_norm'
63      else:
64          batch_norm_str = 'no_batch_norm'
65
66      learning_steps_str = 'learning_steps_'+str(learning_steps)
67      gamma_str = 'gamma_'+str(gamma)
68      auxiliary_str = 'ac_{}_aap_{}_cap_{}'.format(str(float(auxiliary_commission)),
69                                                   str(float(actor_auxiliary_predi
    ction)),
70                                                   str(float(critic_auxiliary_pred
    iction)))
71
72      return 'infer/{}/window_{}/{}/{}/{}/{}/'.format(predictor_type, window_length, batch
    _norm_str,
73                                                      learning_steps_str, gamma_st
    r, auxiliary_str)
74
75
76  def get_variable_scope(window_length, predictor_type, use_batch_norm, learning_s
    teps=0, gamma=0.5,
77                         auxiliary_commission=0, actor_auxiliary_prediction=0, cri
    tic_auxiliary_prediction=0):
78      if use_batch_norm:
79          batch_norm_str = 'batch_norm'
80      else:
81          batch_norm_str = 'no_batch_norm'
82
83      learning_steps_str = 'learning_steps_'+str(learning_steps)
84      gamma_str = 'gamma_'+str(gamma)
85      auxiliary_str = 'ac_{}_aap_{}_cap_{}'.format(str(float(auxiliary_commission)),
86                                                   str(float(actor_auxiliary_predi
    ction)),
87                                                   str(float(critic_auxiliary_pred
    iction)))
88
89      return '{}_window_{}_{}_{}_{}_{}'.format(predictor_type, window_length, batch_no
    rm_str,
90                                               learning_steps_str, gamma_str, auxi
    liary_str)
91
92
93  def stock_predictor_actor(inputs, predictor_type, use_batch_norm, use_previous,
    previous_input,
94                            actor_auxiliary_prediction, target):
95      window_length = inputs.get_shape()[2]
96      assert predictor_type in ['cnn', 'lstm'], 'type must be either cnn or lstm'
97      if predictor_type == 'cnn':
98          net = tflearn.conv_2d(inputs, 32, (1, 3), padding='valid')
99          if use_batch_norm:
100             net = tflearn.layers.normalization.batch_normalization(net)
101         net = tflearn.activations.relu(net)
```

```
102             net = tflearn.conv_2d(net, 32, (1, window_length - 2), padding='valid')
103             if use_batch_norm:
104                 net = tflearn.layers.normalization.batch_normalization(net)
105             net = tflearn.activations.relu(net)
106             if DEBUG:
107                 print('After conv2d:', net.shape)
108 
109             with tf.variable_scope("actor_auxiliary_prediction"+str(target)):
110                 auxiliary_prediction = None
111                 if actor_auxiliary_prediction > 0:
112                     auxiliary_prediction = tflearn.conv_2d(net, 1, (1, 1), padding='
    valid')
113                     auxiliary_prediction = tflearn.flatten(auxiliary_prediction)
114 
115             if use_previous:
116                 net = tflearn.layers.merge_ops.merge([previous_input, net], 'concat',
     axis=-1)
117                 if DEBUG:
118                     print('After concat:', net.shape)
119                 net = tflearn.conv_2d(net, 1, (1, 1), padding='valid')
120                 if DEBUG:
121                     print('After portfolio conv2d:', net.shape)
122             net = tflearn.flatten(net)
123             if DEBUG:
124                 print('Output:', net.shape)
125 
126         elif predictor_type == 'lstm':
127             num_stocks = inputs.get_shape()[1]
128             hidden_dim = 32
129             net = tf.transpose(inputs, [0, 2, 3, 1])
130             resultlist = []
131             reuse = False
132             for i in range(num_stocks):
133                 if i > 0:
134                     reuse = True
135                 print("LAYER:", i)
136                 result = tflearn.layers.lstm(net[:, :, :, i],
137                                              hidden_dim,
138                                              dropout=0.5,
139                                              scope="lstm_actor"+str(target),
140                                              reuse=reuse)
141                 resultlist.append(result)
142             net = tf.stack(resultlist)
143             net = tf.transpose(net, [1, 0, 2])
144             print("STACKED Shape:", net.shape)
145             net = tf.reshape(net, [-1, int(num_stocks), 1, hidden_dim])
146 
147             with tf.variable_scope("actor_auxiliary_prediction"+str(target)):
148                 auxiliary_prediction = None
149                 if actor_auxiliary_prediction > 0:
150                     auxiliary_prediction = tflearn.conv_2d(net, 1, (1, 1), padding='
    valid')
151                     auxiliary_prediction = tflearn.flatten(auxiliary_prediction)
152 
153             if use_previous:
154                 net = tflearn.layers.merge_ops.merge([previous_input, net], 'concat',
     axis=-1)
155                 if DEBUG:
156                     print('After concat:', net.shape)
157                 net = tflearn.conv_2d(net, 1, (1, 1), padding='valid')
158             net = tflearn.flatten(net)
159             if DEBUG:
160                 print('Output:', net.shape)
```

```
161
162     else:
163         raise NotImplementedError
164
165     return net, auxiliary_prediction
166
167 def stock_predictor_critic(inputs, predictor_type, use_batch_norm, use_previous,
     previous_input,
168                            critic_auxiliary_prediction, target):
169     window_length = inputs.get_shape()[2]
170     assert predictor_type in ['cnn', 'lstm'], 'type must be either cnn or lstm'
171     if predictor_type == 'cnn':
172         net = tflearn.conv_2d(inputs, 32, (1, 3), padding='valid')
173         if use_batch_norm:
174             net = tflearn.layers.normalization.batch_normalization(net)
175         net = tflearn.activations.relu(net)
176         net = tflearn.conv_2d(net, 32, (1, window_length - 2), padding='valid')
177         if use_batch_norm:
178             net = tflearn.layers.normalization.batch_normalization(net)
179         net = tflearn.activations.relu(net)
180         if DEBUG:
181             print('After conv2d:', net.shape)
182
183         with tf.variable_scope("critic_auxiliary_prediction"+str(target)):
184             auxiliary_prediction = None
185             if critic_auxiliary_prediction > 0:
186                 auxiliary_prediction = tflearn.conv_2d(net, 1, (1, 1), padding='
    valid')
187                 auxiliary_prediction = tflearn.flatten(auxiliary_prediction)
188
189         if use_previous:
190             net = tflearn.layers.merge_ops.merge([previous_input, net], 'concat',
     axis=-1)
191             if DEBUG:
192                 print('After concat:', net.shape)
193             net = tflearn.conv_2d(net, 1, (1, 1), padding='valid')
194             if DEBUG:
195                 print('After portfolio conv2d:', net.shape)
196         net = tflearn.flatten(net)
197         if DEBUG:
198             print('Output:', net.shape)
199     elif predictor_type == 'lstm':
200         num_stocks = inputs.get_shape()[1]
201         hidden_dim = 32
202         net = tf.transpose(inputs, [0, 2, 3, 1])
203         resultlist = []
204         reuse = False
205         for i in range(num_stocks):
206             if i > 0:
207                 reuse = True
208             print("Layer:", i)
209             result = tflearn.layers.lstm(net[:, :, :, i],
210                                          hidden_dim,
211                                          dropout=0.5,
212                                          scope="lstm_critic"+str(target),
213                                          reuse=reuse)
214             resultlist.append(result)
215         net = tf.stack(resultlist)
216         net = tf.transpose(net, [1, 0, 2])
217         net = tf.reshape(net, [-1, int(num_stocks), 1, hidden_dim])
218
219         with tf.variable_scope("critic_auxiliary_prediction"+str(target)):
220             auxiliary_prediction = None
```

```
221             if critic_auxiliary_prediction > 0:
222                 auxiliary_prediction = tflearn.conv_2d(net, 1, (1, 1), padding='valid')
223                 auxiliary_prediction = tflearn.flatten(auxiliary_prediction)
224 
225         if use_previous:
226             net = tflearn.layers.merge_ops.merge([previous_input, net], 'concat', axis=-1)
227             if DEBUG:
228                 print('After concat:', net.shape)
229             net = tflearn.conv_2d(net, 1, (1, 1), padding='valid')
230         net = tflearn.flatten(net)
231         if DEBUG:
232             print('Output:', net.shape)
233 
234     else:
235         raise NotImplementedError
236 
237     return net, auxiliary_prediction
238 
239 class StockActor(ActorNetwork):
240     def __init__(self, sess, state_dim, action_dim, action_bound, learning_rate, tau, batch_size,
241                  predictor_type, use_batch_norm, use_previous=False, auxiliary_commission=0,
242                  actor_auxiliary_prediction=0):
243         self.predictor_type = predictor_type
244         self.use_batch_norm = use_batch_norm
245         self.use_previous = use_previous
246         self.auxiliary_commission = auxiliary_commission
247         self.actor_auxiliary_prediction = actor_auxiliary_prediction
248         ActorNetwork.__init__(self, sess, state_dim, action_dim, action_bound, learning_rate, tau, batch_size)
249 
250     def create_actor_network(self, target):
251         """
252     self.s_dim: a list specifies shape
253     """
254         nb_classes, window_length = self.s_dim
255         assert nb_classes == self.a_dim[0]
256         assert window_length > 2, 'This architecture only support window length larger than 2.'
257         inputs = tflearn.input_data(shape=[None] + self.s_dim + [1], name='input')
258 
259         portfolio_inputs = None
260         portfolio_reshaped = None
261         if self.use_previous:
262             portfolio_inputs = tflearn.input_data(shape=[None] + self.a_dim, name='portfolio_input')
263             portfolio_reshaped = tflearn.reshape(portfolio_inputs, new_shape=[-1]+self.a_dim+[1, 1])
264 
265         net, auxil = stock_predictor_actor(inputs, self.predictor_type, self.use_batch_norm,
266                                            self.use_previous, portfolio_reshaped, self.actor_auxiliary_prediction,
267                                            target)
268         out = tf.nn.softmax(net)
269         scaled_out = tf.multiply(out, self.action_bound)
270 
271         loss = None
272         future_y_inputs = None
273         if self.actor_auxiliary_prediction > 0:
```

```
274             future_y_inputs = tflearn.input_data(shape=[None] + self.a_dim, name
    ='portfolio_input')
275             loss = self.actor_auxiliary_prediction* \
276                 tf.reduce_mean(tf.reduce_sum(tf.square(auxil - future_y_inputs),
     axis=-1))
277 
278         return inputs, out, scaled_out, portfolio_inputs, loss, future_y_inputs
279 
280     def train(self, inputs, a_gradient, portfolio_inputs=None, future_y_inputs=N
    one):
281         window_length = self.s_dim[1]
282         inputs = inputs[:, :, -window_length:, :]
283         if not self.use_previous:
284             self.sess.run([self.optimize], feed_dict={
285                 self.inputs: inputs,
286                 self.action_gradient: a_gradient
287             })
288         else:
289             if self.actor_auxiliary_prediction > 0 and self.auxiliary_commission
    :
290                 self.sess.run([self.optimize, self.optimize_comm, self.optimize_
    prediction], feed_dict={
291                     self.inputs: inputs,
292                     self.portfolio_inputs: portfolio_inputs,
293                     self.action_gradient: a_gradient,
294                     self.future_y_inputs: future_y_inputs
295                 })
296             elif self.actor_auxiliary_prediction > 0:
297                 self.sess.run([self.optimize, self.optimize_prediction], feed_di
    ct={
298                     self.inputs: inputs,
299                     self.portfolio_inputs: portfolio_inputs,
300                     self.action_gradient: a_gradient,
301                     self.future_y_inputs: future_y_inputs
302                 })
303             elif self.auxiliary_commission > 0:
304                 self.sess.run([self.optimize, self.optimize_comm], feed_dict={
305                     self.inputs: inputs,
306                     self.portfolio_inputs: portfolio_inputs,
307                     self.action_gradient: a_gradient
308                 })
309             else:
310                 self.sess.run([self.optimize], feed_dict={
311                     self.inputs: inputs,
312                     self.portfolio_inputs: portfolio_inputs,
313                     self.action_gradient: a_gradient
314                 })
315 
316     def predict(self, inputs, portfolio_inputs=None):
317         window_length = self.s_dim[1]
318         inputs = inputs[:, :, -window_length:, :]
319         if not self.use_previous:
320             return self.sess.run(self.scaled_out, feed_dict={
321                 self.inputs: inputs
322             })
323         else:
324             return self.sess.run(self.scaled_out, feed_dict={
325                 self.inputs:inputs,
326                 self.portfolio_inputs: portfolio_inputs
327             })
328 
329     def predict_target(self, inputs, portfolio_inputs=None):
330         window_length = self.s_dim[1]
```

```
331         inputs = inputs[:, :, -window_length:, :]
332         if not self.use_previous:
333             return self.sess.run(self.target_scaled_out, feed_dict={
334                 self.target_inputs: inputs
335             })
336         else:
337             return self.sess.run(self.target_scaled_out, feed_dict={
338                 self.target_inputs: inputs,
339                 self.target_portfolio_inputs: portfolio_inputs
340             })
341
342
343 class StockCritic(CriticNetwork):
344     def __init__(self, sess, state_dim, action_dim, learning_rate, tau, num_acto
    r_vars,
345                  predictor_type, use_batch_norm, use_previous=False, critic_auxi
    liary_prediction=0):
346         self.predictor_type = predictor_type
347         self.use_batch_norm = use_batch_norm
348         self.use_previous = use_previous
349         self.critic_auxiliary_prediction = critic_auxiliary_prediction
350         CriticNetwork.__init__(self, sess, state_dim, action_dim, learning_rate,
     tau, num_actor_vars)
351
352     def create_critic_network(self, target):
353         inputs = tflearn.input_data(shape=[None] + self.s_dim + [1])
354         action = tflearn.input_data(shape=[None] + self.a_dim)
355
356         portfolio_inputs = None
357         portfolio_reshaped = None
358         if self.use_previous:
359             portfolio_inputs = tflearn.input_data(shape=[None] + self.a_dim, nam
    e='portfolio_input')
360             portfolio_reshaped = tflearn.reshape(portfolio_inputs, new_shape=[-1
    ]+self.a_dim+[1, 1])
361
362         net, auxil = stock_predictor_critic(inputs, self.predictor_type, self.us
    e_batch_norm,
363                                             self.use_previous, portfolio_reshape
    d, self.critic_auxiliary_prediction,
364                                             target)
365
366         loss = 0
367         future_y_inputs = None
368         if self.critic_auxiliary_prediction > 0:
369             future_y_inputs = tflearn.input_data(shape=[None] + self.a_dim, name
    ='portfolio_input')
370             loss = self.critic_auxiliary_prediction* \
371                 tf.reduce_mean(tf.reduce_sum(tf.square(auxil - future_y_inputs),
     axis=-1))
372
373         # Add the action tensor in the 2nd hidden layer
374         # Use two temp layers to get the corresponding weights and biases
375         t1 = tflearn.fully_connected(net, 64)
376         t2 = tflearn.fully_connected(action, 64)
377
378         net = tf.add(t1, t2)
379         if self.use_batch_norm:
380             net = tflearn.layers.normalization.batch_normalization(net)
381         net = tflearn.activations.relu(net)
382
383         # linear layer connected to 1 output representing Q(s,a)
384         # Weights are init to Uniform[-3e-3, 3e-3]
```

```
385         w_init = tflearn.initializations.uniform(minval=-0.003, maxval=0.003)
386         out = tflearn.fully_connected(net, 1, weights_init=w_init)
387         return inputs, action, out, portfolio_inputs, loss, future_y_inputs
388 
389     def train(self, inputs, action, predicted_q_value, portfolio_inputs=None, fu
    ture_y_inputs=None):
390         window_length = self.s_dim[1]
391         inputs = inputs[:, :, -window_length:, :]
392         if not self.use_previous:
393             return self.sess.run([self.out, self.optimize], feed_dict={
394                 self.inputs: inputs,
395                 self.action: action,
396                 self.predicted_q_value: predicted_q_value
397             })
398         else:
399             if self.critic_auxiliary_prediction > 0:
400                 return self.sess.run([self.out, self.optimize], feed_dict={
401                     self.inputs: inputs,
402                     self.portfolio_inputs: portfolio_inputs,
403                     self.action: action,
404                     self.predicted_q_value: predicted_q_value,
405                     self.future_y_inputs: future_y_inputs
406                 })
407 
408             else:
409                 return self.sess.run([self.out, self.optimize], feed_dict={
410                     self.inputs: inputs,
411                     self.portfolio_inputs: portfolio_inputs,
412                     self.action: action,
413                     self.predicted_q_value: predicted_q_value
414                 })
415 
416     def predict(self, inputs, action, portfolio_inputs=None):
417         window_length = self.s_dim[1]
418         inputs = inputs[:, :, -window_length:, :]
419         if not self.use_previous:
420             return self.sess.run(self.out, feed_dict={
421                 self.inputs: inputs,
422                 self.action: action
423             })
424         else:
425             return self.sess.run(self.out, feed_dict={
426                 self.inputs: inputs,
427                 self.portfolio_inputs: portfolio_inputs,
428                 self.action: action
429             })
430 
431     def predict_target(self, inputs, action, portfolio_inputs=None):
432         window_length = self.s_dim[1]
433         inputs = inputs[:, :, -window_length:, :]
434         if not self.use_previous:
435             return self.sess.run(self.target_out, feed_dict={
436                 self.target_inputs: inputs,
437                 self.target_action: action
438             })
439         else:
440             return self.sess.run(self.target_out, feed_dict={
441                 self.target_inputs: inputs,
442                 self.target_portfolio_inputs: portfolio_inputs,
443                 self.target_action: action
444             })
445 
446     def action_gradients(self, inputs, actions, portfolio_inputs=None):
```

```
447             window_length = self.s_dim[1]
448             inputs = inputs[:, :, -window_length:, :]
449             if not self.use_previous:
450                 return self.sess.run(self.action_grads, feed_dict={
451                     self.inputs: inputs,
452                     self.action: actions
453                 })
454             else:
455                 return self.sess.run(self.action_grads, feed_dict={
456                     self.inputs: inputs,
457                     self.portfolio_inputs: portfolio_inputs,
458                     self.action: actions
459                 })
460
461
462 def obs_normalizer(observation):
463     """ Preprocess observation obtained by environment
464
465     Args:
466         observation: (nb_classes, window_length, num_features) or with info
467
468     Returns: normalized
469
470     """
471         if isinstance(observation, tuple):
472             observation = observation[0]
473         # directly use close/open ratio as feature
474         observation = observation[:, :, 3:4] / observation[:, :, 0:1]
475         observation = normalize(observation)
476         return observation
477
478
479 def test_model(env, model):
480     observation, info = env.reset()
481     done = False
482     while not done:
483         action = model.predict_single(observation)
484         observation, _, done, _ = env.step(action)
485     env.render()
486
487
488 def test_model_multiple(env, models):
489     observation, info = env.reset()
490     done = False
491     while not done:
492         observation, weights = observation['obs'], observation['weights']
493         actions = []
494         for i, model in enumerate(models):
495             model_obs = {'obs': observation, 'weights': weights[i]}
496             actions.append(model.predict_single(model_obs))
497         actions = np.array(actions)
498         observation, _, done, info = env.step(actions)
499     # env.render()
500
501
502 if __name__ == '__main__':
503
504     parser = argparse.ArgumentParser(description='Provide arguments for training different DD
    PG models')
505
506     parser.add_argument('--debug', '-d', help='print debug statement', default=False, t
    ype=bool)
507     parser.add_argument('--predictor_type', '-p', help='cnn or lstm predictor', required=Tr
```

```
ue)
    parser.add_argument('--window_length', '-w', help='observation window length', required=True, type=int)
    parser.add_argument('--batch_norm', '-b', help='whether to use batch normalization', required=True, type=bool)
    parser.add_argument('--learning_steps', '-l', help='number of learning steps for DDPG', required=True, type=int)
    parser.add_argument('--auxil_commission', '-ac', help='whether to use auxiliary commission', default=0, type=float)
    parser.add_argument('--actor_auxil_prediction', '-aap', help='whether to use actor auxiliary prediction', default=0, type=float)
    parser.add_argument('--critic_auxil_prediction', '-ap', help='whether to use critic_auxiliary prediction', default=0, type=float)
    parser.add_argument('--actor_tau', '-at', help='actor tau constant', default=1e-3, type=float)
    parser.add_argument('--critic_tau', '-ct', help='critic tau constant', default=1e-3, type=float)
    parser.add_argument('--actor_learning_rate', '-al', help='actor learning rate', default=1e-4, type=float)
    parser.add_argument('--critic_learning_rate', '-cl', help='critic learning rate', default=1e-3, type=float)
    parser.add_argument('--batch_size', '-bs', help='batch size', default=64, type=int)
    parser.add_argument('--action_bound', '-ab', help='action bound', default=1, type=int)
    parser.add_argument('--load_weights', '-lw', help='load previous weights', default=False, type=bool)
    parser.add_argument('--gamma', '-g', help='gamma value', default=0.5, type=float)
    parser.add_argument('--training_episodes', '-te', help='number of episodes to train on', default=600, type=int)
    parser.add_argument('--max_rollout_steps', '-mre', help='number of steps to rollout in an episode', default=1000, type=int)
    parser.add_argument('--buffer_size', '-bus', help='replay buffer size', default=100000, type=int)
    parser.add_argument('--seed', '-s', help='seed value', default=1337, type=int)

    args = vars(parser.parse_args())

    pprint.pprint(args)

    DEBUG=args['debug']
    predictor_type = args['predictor_type']
    window_length = args['window_length']
    use_batch_norm = args['batch_norm']
    learning_steps = args['learning_steps']
    auxil_commission = args['auxil_commission']
    actor_auxil_prediction = args['actor_auxil_prediction']
    critic_auxil_prediction = args['critic_auxil_prediction']
    actor_tau = args['actor_tau']
    critic_tau = args['critic_tau']
    actor_learning_rate = args['actor_learning_rate']
    critic_learning_rate = args['critic_learning_rate']
    batch_size = args['batch_size']
    action_bound = args['action_bound']
    load_weights = args['load_weights']
    gamma = args['gamma']
    training_episodes = args['training_episodes']
    max_rollout_steps = args['max_rollout_steps']
    buffer_size = args['buffer_size']
    seed = args['seed']

    assert args['predictor_type'] in ['cnn', 'lstm'], 'Predictor must be either cnn or lstm'

```

```
554 ################################### NASDAQ ###################################
    #######
555
556     history, abbreviation = read_stock_history(filepath='utils/datasets/stocks_history_targe
    t.h5')
557     history = history[:, :, :4]
558     target_stocks = abbreviation
559     num_training_time = 1095
560
561     # get target history
562     target_history = np.empty(shape=(len(target_stocks), num_training_time, hist
    ory.shape[2]))
563     for i, stock in enumerate(target_stocks):
564         target_history[i] = history[abbreviation.index(stock), :num_training_tim
    e, :]
565     print("target:", target_history.shape)
566
567     testing_stocks = abbreviation
568     test_history = np.empty(shape=(len(testing_stocks), history.shape[1] - num_t
    raining_time,
569                                    history.shape[2]))
570     for i, stock in enumerate(testing_stocks):
571         test_history[i] = history[abbreviation.index(stock), num_training_time:,
     :]
572     print("test:", test_history.shape)
573
574     train_env = PortfolioEnv(target_history,
575                              target_stocks,
576                              steps=min(max_rollout_steps, target_history.shape[1
    ]-window_length-learning_steps-1),
577                              window_length=window_length)
578     infer_train_env = PortfolioEnv(target_history,
579                                    target_stocks,
580                                    steps=target_history.shape[1]-window_length-l
    earning_steps-1,
581                                    window_length=window_length)
582     infer_test_env = PortfolioEnv(test_history,
583                                   testing_stocks,
584                                   steps=test_history.shape[1]-window_length-lear
    ning_steps-1,
585                                   window_length=window_length)
586     infer_train_env.reset()
587     infer_test_env.reset()
588     nb_classes = len(target_stocks) + 1
589
590     action_dim = [nb_classes]
591     state_dim = [nb_classes, window_length]
592
593     actor_noise = OrnsteinUhlenbeckActionNoise(mu=np.zeros(action_dim))
594     model_save_path = get_model_path(window_length, predictor_type, use_batch_no
    rm,
595                                      learning_steps, gamma, auxil_commission, ac
    tor_auxil_prediction,
596                                      critic_auxil_prediction)
597     summary_path = get_result_path(window_length, predictor_type, use_batch_norm
    ,
598                                    learning_steps, gamma, auxil_commission, acto
    r_auxil_prediction,
599                                    critic_auxil_prediction)
600     infer_path = get_infer_path(window_length, predictor_type, use_batch_norm,
601                                 learning_steps, gamma, auxil_commission, actor_a
    uxil_prediction,
602                                 critic_auxil_prediction)
```

```
603     variable_scope = get_variable_scope(window_length, predictor_type, use_batch
    _norm,
604                                         learning_steps, gamma, auxil_commission,
     actor_auxil_prediction,
605                                         critic_auxil_prediction)
606
607     with tf.variable_scope(variable_scope):
608         sess = tf.Session()
609         actor = StockActor(sess=sess, state_dim=state_dim, action_dim=action_dim
    , action_bound=action_bound,
610                            learning_rate=1e-4, tau=actor_tau, batch_size=batch_s
    ize,
611                            predictor_type=predictor_type, use_batch_norm=use_bat
    ch_norm, use_previous=True,
612                            auxiliary_commission=auxil_commission, actor_auxiliar
    y_prediction=actor_auxil_prediction)
613         critic = StockCritic(sess=sess, state_dim=state_dim, action_dim=action_d
    im, tau=critic_tau,
614                              learning_rate=1e-3, num_actor_vars=actor.get_num_tr
    ainable_vars(),
615                              predictor_type=predictor_type, use_batch_norm=use_b
    atch_norm, use_previous=True,
616                              critic_auxiliary_prediction=critic_auxil_prediction
    )
617         ddpg_model = DDPG(train_env, sess, actor, critic, actor_noise, obs_norma
    lizer=obs_normalizer,
618                           gamma=gamma, training_episodes=training_episodes, max_
    rollout_steps=max_rollout_steps,
619                           buffer_size=buffer_size, seed=seed, batch_size=batch_s
    ize, model_save_path=model_save_path,
620                           summary_path=summary_path, infer_path=infer_path, infe
    r_train_env=infer_train_env,
621                           infer_test_env=infer_test_env, learning_steps=learning
    _steps)
622         ddpg_model.initialize(load_weights=load_weights, verbose=False)
623         ddpg_model.train()
```

portfolio.py

```
"""
Modified from https://github.com/wassname/rl-portfolio-management/blob/master/src/environments/portfolio.py
Modified from https://github.com/vermouth1992/drl-portfolio-management/blob/master/src/environment/portfolio.py
"""
from __future__ import print_function

from pprint import pprint

import matplotlib
matplotlib.use('Agg')
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import gym
import gym.spaces

from utils.data import date_to_index, index_to_date, index_to_date_offset
from pgportfolio.tools.configprocess import load_config

eps = 1e-8


def random_shift(x, fraction):
    """ Apply a random shift to a pandas series. """
    min_x, max_x = np.min(x), np.max(x)
    m = np.random.uniform(-fraction, fraction, size=x.shape) + 1
    return np.clip(x * m, min_x, max_x)


def scale_to_start(x):
    """ Scale pandas series so that it starts at one. """
    x = (x + eps) / (x[0] + eps)
    return x


def sharpe(returns, freq=30, rfr=0):
    """ Given a set of returns, calculates naive (rfr=0) sharpe (eq 28). """
    return (np.sqrt(freq) * np.mean(returns - rfr + eps)) / np.std(returns - rfr + eps)


def max_drawdown(returns):
    """ Max drawdown. See https://www.investopedia.com/terms/m/maximum-drawdown-mdd.asp """
    peak = returns.max()
    trough = returns[returns.argmax():].min()
    return (trough - peak) / (peak + eps)


class DataGenerator(object):
    """Acts as data provider for each new episode."""

    def __init__(self, history, abbreviation, steps=730, window_length=50, start_idx=0, start_date=None):
        """

    Args:
        history: (num_stocks, timestamp, 5) open, high, low, close, volume
        abbreviation: a list of length num_stocks with assets name
        steps: the total number of steps to simulate, default is 2 years
        window_length: observation window, must be less than 50
        start_date: the date to start. Default is None and random pick one.
                It should be a string e.g. '2012-08-13'
```

```
    """
        assert history.shape[0] == len(abbreviation), 'Number of stock is not consistent'
        import copy

        self.steps = steps + 1
        self.window_length = window_length
        self.start_idx = start_idx
        self.start_date = start_date

        # make immutable class
        self._data = history.copy()  # all data
        self.asset_names = copy.copy(abbreviation)

    def _step(self):
        # get observation matrix from history, exclude volume, maybe volume is useful as it
        # indicates how market total investment changes. Normalize could be critical here
        self.step += 1
        obs = self.data[:, self.step:self.step + self.window_length, :].copy()
        # normalize obs with open price

        # used for compute optimal action and sanity check
        ground_truth_obs = self.data[:, self.step + self.window_length:self.step + self.window_length + 1, :].copy()

        done = self.step >= self.steps
        return obs, done, ground_truth_obs

    def reset(self):
        self.step = 0

        # get data for this episode, each episode might be different.
        if self.start_date is None:
            print("LOW:", self.window_length)
            print("HIGH:", self._data.shape[1] - self.steps)
            self.idx = np.random.randint(
                low=self.window_length, high=self._data.shape[1] - self.steps)
        else:
            # compute index corresponding to start_date for repeatable sequence
            self.idx = date_to_index(self.start_date) - self.start_idx
            assert self.idx >= self.window_length and self.idx <= self._data.shape[1] - self.steps, \
                'Invalid start date, must be window_length day after start date and simulation steps day before end date'
        # print('Start date: {}'.format(index_to_date(self.idx)))
        data = self._data[:, self.idx - self.window_length:self.idx + self.steps + 1, :4]
        # apply augmentation?
        self.data = data
        return self.data[:, self.step:self.step + self.window_length, :].copy(), \
               self.data[:, self.step + self.window_length:self.step + self.window_length + 1, :].copy()


class PortfolioSim(object):
    """
  Portfolio management sim.
  Params:
  - cost e.g. 0.0025 is max in Poliniex
  Based off [Jiang 2017](https://arxiv.org/abs/1706.10059)
  """
```

```
117
118     def __init__(self, asset_names=list(), steps=730, trading_cost=0.0025, time_
    cost=0.0):
119         self.asset_names = asset_names
120         self.cost = trading_cost
121         self.time_cost = time_cost
122         self.steps = steps
123
124     def _step(self, w1, y1):
125         w0 = self.w0 #old_weights'
126         p0 = self.p0 #p'
127
128         c1 = self.cost * (np.abs(w0[1:] - w1[1:])).sum()
129         p1 = p0 * (1 - c1) * np.dot(y1, w1) # p_(t+1)''
130
131         dw1 = (y1 * w1) / (np.dot(y1, w1) + eps)  # (eq7) weights evolve into
132
133         # can't have negative holdings in this model (no shorts)
134         p1 = np.clip(p1, 0, np.inf)
135
136         rho1 = p1 / p0 - 1  # rate of returns
137         r1 = np.log((p1 + eps) / (p0 + eps))  # (eq10) log rate of return
138         # (eq22) immediate reward is log rate of return scaled by episode length
139         reward = r1 / self.steps * 1000
140
141         # remember for next step
142         self.w0 = dw1
143         self.p0 = p1
144
145         # if we run out of money, we're done
146         done = bool(p1 == 0)
147
148         # should only return single values, not list
149         info = {
150             "reward": reward,
151             "log_return": r1,
152             "portfolio_value": p1,
153             "return": y1[1:].mean(),
154             "rate_of_return": rho1,
155             "weights_mean": w1.mean(),
156             "weights_std": w1.std(),
157             "cost": p0*c1,
158             "weights": w1,
159             "evolved_weights": dw1
160         }
161
162         self.infos.append(info)
163         return reward, info, done
164
165     def reset(self):
166         self.infos = []
167         self.p0 = 1.0
168         self.w0 = np.zeros(len(self.asset_names) + 1)
169         self.w0[0] = 1
170
171
172 class PortfolioEnv(gym.Env):
173     """
174     An environment for financial portfolio management.
175     Financial portfolio management is the process of constant redistribution of a fund into different
176     financial products.
177     Based on [Jiang 2017](https://arxiv.org/abs/1706.10059)
178     """
```

```
179
180     metadata = {'render.modes': ['human', 'ansi']}
181
182     def __init__(self,
183                  history,
184                  abbreviation,
185                  steps=730,  # 2 years
186                  trading_cost=0.0025,
187                  time_cost=0.00,
188                  window_length=50,
189                  start_idx=0,
190                  sample_start_date=None,
191                  seed=31415
192                  ):
193         """
194     An environment for financial portfolio management.
195     Params:
196       steps - steps in episode
197       scale - scale data and each episode (except return)
198       augment - fraction to randomly shift data by
199       trading_cost - cost of trade as a fraction
200       time_cost - cost of holding as a fraction
201       window_length - how many past observations to return
202       start_idx - The number of days from '2012-08-13' of the dataset
203       sample_start_date - The start date sampling from the history
204     """
205         plt.rcParams["figure.figsize"] = (15,8)
206         np.random.seed(seed)
207         self.window_length = window_length
208         self.num_stocks = history.shape[0]
209         self.start_idx = start_idx
210         self.steps = steps
211
212         self.src = DataGenerator(history, abbreviation, steps=steps, window_leng
    th=window_length, start_idx=start_idx,
213                                  start_date=sample_start_date)
214
215         self.sim = PortfolioSim(
216             asset_names=abbreviation,
217             trading_cost=trading_cost,
218             time_cost=time_cost,
219             steps=steps)
220
221         # openai gym attributes
222         # action will be the portfolio weights from 0 to 1 for each asset
223         self.action_space = gym.spaces.Box(
224             0, 1, shape=len(self.src.asset_names) + 1)  # include cash
225
226         # get the observation space from the data min and max
227         self.observation_space = gym.spaces.Box(low=-np.inf, high=np.inf, shape=
    (len(abbreviation) + 1, window_length,
228
     1))
229
230     def _step(self, action):
231         """
232     Step the env.
233     Actions should be portfolio [w0...]
234     - Where wn is a portfolio weight from 0 to 1. The first is cash_bias
235     - cn is the portfolio conversion weights see PortioSim._step for description
236     """
237         np.testing.assert_almost_equal(
238             action.shape,
```

```
239             (len(self.sim.asset_names) + 1,)
240         )
241
242         # normalise just in case
243         action = np.clip(action, 0, 1)
244
245         weights = action  # np.array([cash_bias] + list(action))  # [w0, w1...]
246         weights /= (weights.sum() + eps)
247         weights[0] += np.clip(1 - weights.sum(), 0, 1)  # so if weights are all
    zeros we normalise to [1,0...]
248
249         assert ((action >= 0) * (action <= 1)).all(), 'all action values should be between 0
     and 1. Not %s' % action
250         np.testing.assert_almost_equal(
251             np.sum(weights), 1.0, 3, err_msg='weights should sum to 1. action="%s"' % weig
    hts)
252
253         observation, done1, ground_truth_obs = self.src._step()
254
255         # concatenate observation with ones
256         cash_observation = np.ones((1, self.window_length, observation.shape[2])
    )
257         observation = np.concatenate((cash_observation, observation), axis=0)
258
259         cash_ground_truth = np.ones((1, 1, ground_truth_obs.shape[2]))
260         ground_truth_obs = np.concatenate((cash_ground_truth, ground_truth_obs),
     axis=0)
261
262         # relative price vector of last observation day (close/open)
263         close_price_vector = observation[:, -1, 3]
264         open_price_vector = observation[:, -1, 0]
265         #open_price_vector = observation[:, -2, 3]
266         y1 = close_price_vector / open_price_vector
267         reward, info, done2 = self.sim._step(weights, y1)
268
269         # calculate return for buy and hold a bit of each asset
270         info['market_value'] = np.cumprod([inf["return"] for inf in self.infos + [in
    fo]])[-1]
271         info['open_prices'] = open_price_vector
272         # add dates
273         info['date'] = index_to_date(self.start_idx + self.src.idx + self.src.ste
    p)
274         info['steps'] = self.src.step
275         info['next_obs'] = ground_truth_obs
276         info['next_y1'] = ground_truth_obs[:, -1, 3] / ground_truth_obs[:, -1, 0]
277
278         self.infos.append(info)
279
280         observation = {'obs': observation, 'weights':  self.sim.w0}
281
282         return observation, reward, done1 or done2, info
283
284     def _reset(self):
285         self.infos = []
286         self.sim.reset()
287         observation, ground_truth_obs = self.src.reset()
288         cash_observation = np.ones((1, self.window_length, observation.shape[2])
    )
289         observation = np.concatenate((cash_observation, observation), axis=0)
290         cash_ground_truth = np.ones((1, 1, ground_truth_obs.shape[2]))
291         ground_truth_obs = np.concatenate((cash_ground_truth, ground_truth_obs),
     axis=0)
292         info = {}
```

```
293         info['next_obs'] = ground_truth_obs
294 
295         observation = {'obs': observation, 'weights': self.sim.w0}
296         return observation, info
297 
298     def _render(self, mode='human', close=False):
299         if close:
300             return
301         if mode == 'ansi':
302             pprint(self.infos[-1])
303         elif mode == 'human':
304             self.plot()
305 
306     def plot(self):
307         #print("HERE")
308         # show a plot of portfolio vs mean market performance
309         fig, axes = plt.subplots(nrows=4, ncols=1)
310         df_info = pd.DataFrame(self.infos)
311         df_info.index = df_info["date"]
312         mdd = max_drawdown(df_info.rate_of_return + 1)
313         sharpe_ratio = sharpe(df_info.rate_of_return)
314         title = 'max_drawdown={: 2.2%} sharpe_ratio={: 2.4f}'.format(mdd, sharpe_ratio)
315         df_info[["portfolio_value", "market_value"]].plot(title=title, ax=axes[0], rot=
    30)
316 
317         prices = [info["open_prices"] for info in self.infos]
318         prices = np.array(prices)
319         axes[1].set_ylabel('Prices')
320         for ind in range(prices.shape[1]):
321             axes[1].plot(prices[:, ind])
322 
323         allocations = [info["weights"] for info in self.infos]
324         allocations = np.array(allocations)
325         axes[2].set_ylabel('Action')
326         for ind in range(allocations.shape[1]):
327             axes[2].plot(allocations[:, ind])
328 
329         costs = [info["cost"] for info in self.infos]
330         costs = np.cumsum(costs).flatten()
331         axes[3].set_ylabel('Cost')
332         axes[3].plot(costs)
333         plt.show()
334 
335 
336     def plot_costs(self):
337         costs = [info["cost"] for info in self.infos]
338         costs = np.array(costs)
339         plt.plot(costs)
340 
341 class MultiActionPortfolioEnv(PortfolioEnv):
342     def __init__(self,
343                  history,
344                  abbreviation,
345                  model_names,
346                  steps=730,  # 2 years
347                  trading_cost=0.0025,
348                  time_cost=0.00,
349                  window_length=50,
350                  start_idx=0,
351                  sample_start_date=None,
352                  offset=1095
353                  ):
354         super(MultiActionPortfolioEnv, self).__init__(history, abbreviation, ste
```

```
    ps, trading_cost, time_cost, window_length,
                                  start_idx, sample_start_date)
        self.model_names = model_names
        self.offset = offset
        # need to create each simulator for each model
        self.sim = [PortfolioSim(
            asset_names=abbreviation,
            trading_cost=trading_cost,
            time_cost=time_cost,
            steps=steps) for _ in range(len(self.model_names))]

    def _step(self, action):
        """ Step the environment by a vector of actions

    Args:
      action: (num_models, num_stocks + 1)

    Returns:

    """
        assert action.ndim == 2, 'Action must be a two dimensional array with shape (num_models, n
    um_stocks + 1)'
        assert action.shape[1] == len(self.sim[0].asset_names) + 1
        assert action.shape[0] == len(self.model_names)
        # normalise just in case
        action = np.clip(action, 0, 1)
        weights = action  # np.array([cash_bias] + list(action))  # [w0, w1...]
        weights /= (np.sum(weights, axis=1, keepdims=True) + eps)
        # so if weights are all zeros we normalise to [1,0...]
        weights[:, 0] += np.clip(1 - np.sum(weights, axis=1), 0, 1)
        assert ((action >= 0) * (action <= 1)).all(), 'all action values should be between 0
     and 1. Not %s' % action
        np.testing.assert_almost_equal(np.sum(weights, axis=1), np.ones(shape=(w
    eights.shape[0])), 3,
                                       err_msg='weights should sum to 1. action="%s"' % we
    ights)
        observation, done1, ground_truth_obs = self.src._step()

        # concatenate observation with ones
        cash_observation = np.ones((1, self.window_length, observation.shape[2])
    )
        observation = np.concatenate((cash_observation, observation), axis=0)

        cash_ground_truth = np.ones((1, 1, ground_truth_obs.shape[2]))
        ground_truth_obs = np.concatenate((cash_ground_truth, ground_truth_obs),
     axis=0)

        # relative price vector of last observation day (close/open)
        close_price_vector = observation[:, -1, 3]
        open_price_vector = observation[:, -1, 0]
        y1 = close_price_vector / open_price_vector

        rewards = np.empty(shape=(weights.shape[0]))
        info = {}
        rate_of_returns = {}
        dones = np.empty(shape=(weights.shape[0]), dtype=bool)
        for i in range(weights.shape[0]):
            reward, current_info, done2 = self.sim[i]._step(weights[i], y1)
            rewards[i] = reward
            info[self.model_names[i]] = current_info['portfolio_value']
            info['return'] = current_info['return']
            rate_of_returns[self.model_names[i]] = current_info['rate_of_return']
            dones[i] = done2
```

```
411
412         # calculate return for buy and hold a bit of each asset
413         info['market_value'] = np.cumprod([inf["return"] for inf in self.infos + [in
    fo]])[-1]
414         # add dates
415         info['date'] = index_to_date_offset(self.start_idx + self.src.idx + self.
    src.step, self.offset)
416         info['steps'] = self.src.step
417         info['next_obs'] = ground_truth_obs
418
419         self.infos.append(info)
420         self.rate_of_returns.append(rate_of_returns)
421
422         observation = {'obs': observation, 'weights': np.array([self.sim[i].w0 for
     i in range(weights.shape[0])])}
423         return observation, rewards, np.all(dones) or done1, info
424
425     def _reset(self):
426         self.infos = []
427         self.rate_of_returns = []
428         for sim in self.sim:
429             sim.reset()
430         observation, ground_truth_obs = self.src.reset()
431         cash_observation = np.ones((1, self.window_length, observation.shape[2])
    )
432         observation = np.concatenate((cash_observation, observation), axis=0)
433         cash_ground_truth = np.ones((1, 1, ground_truth_obs.shape[2]))
434         ground_truth_obs = np.concatenate((cash_ground_truth, ground_truth_obs),
     axis=0)
435         info = {}
436         info['next_obs'] = ground_truth_obs
437
438         observation = {'obs': observation, 'weights': np.array([sim.w0 for sim in
    self.sim])}
439         return observation, info
440
441     def make_df(self):
442         self.df_info = pd.DataFrame(self.infos)
443
444     def plot(self):
445         df_info = self.df_info
446         df_info.index = df_info["date"]
447         fig = plt.gcf()
448         title = 'Trading Performance of Various Models'
449         # for model_name in self.model_names:
450         #     df_info[[model_name]].plot(title=title, fig=fig, rot=30)
451         df_info[self.model_names + ['market_value']].plot(title=title, fig=fig, ro
    t=30)
452         plt.ylabel('Cumulative Wealth')
453         plt.grid()
454
455     def stats(self):
456         stats = {}
457         for model_name in self.model_names:
458             dic = {}
459             dic['fAPV'] = self.infos[-1][model_name]
460             model_returns = [rate_of_return[model_name] for rate_of_return in se
    lf.rate_of_returns]
461             dic['sharpe'] = sharpe(np.array(model_returns))
462             dic['mdd'] = max_drawdown(np.array(model_returns)+1)
463             stats[model_name] = dic
464         return stats
```

```
"""
Modified from https://github.com/vermouth1992/drl-portfolio-management/blob/master/src/model/ddpg/actor.py
"""

import tensorflow as tf


# ===========================
#   Actor DNNs
# ===========================

class ActorNetwork(object):
    def __init__(self, sess, state_dim, action_dim, action_bound, learning_rate, tau, batch_size):
        """

    Args:
        sess: a tensorflow session
        state_dim: a list specifies shape
        action_dim: a list specified action shape
        action_bound: whether to normalize action in the end
        learning_rate: learning rate
        tau: target network update parameter
        batch_size: use for normalization
    """
        self.sess = sess
        assert isinstance(state_dim, list), 'state_dim must be a list.'
        self.s_dim = state_dim
        assert isinstance(action_dim, list), 'action_dim must be a list.'
        self.a_dim = action_dim
        self.action_bound = action_bound
        self.learning_rate = learning_rate
        self.tau = tau
        self.batch_size = batch_size

        # Actor Network
        self.inputs, self.out, self.scaled_out, self.portfolio_inputs, \
            self.auxil_loss, self.future_y_inputs = self.create_actor_network(False)

        self.network_params = tf.trainable_variables()

        # Target Network
        self.target_inputs, self.target_out, self.target_scaled_out, \
            self.target_portfolio_inputs, self.target_auxil_loss, self.target_future_y_inputs \
            = self.create_actor_network(True)

        self.target_network_params = tf.trainable_variables()[
                                     len(self.network_params):]

        # Op for periodically updating target network with online network
        # weights
        self.update_target_network_params = \
            [self.target_network_params[i].assign(tf.multiply(self.network_params[i], self.tau) +
                                                  tf.multiply(self.target_network_params[i], 1. - self.tau))
             for i in range(len(self.target_network_params))]

        # This gradient will be provided by the critic network
        self.action_gradient = tf.placeholder(tf.float32, [None] + self.a_dim)

```

```
59          optimizer = tf.train.AdamOptimizer(self.learning_rate)
60
61          actor_grad_params = [v for v in self.network_params if "actor_auxiliary_predict
    ionFalse" not in v.name]
62          # Combine the gradients here
63          self.unnormalized_actor_gradients = tf.gradients(
64              self.scaled_out, actor_grad_params, -self.action_gradient)
65
66          self.actor_gradients = list(map(lambda x: tf.div(x, self.batch_size), se
    lf.unnormalized_actor_gradients))
67
68          # Optimization Op
69          self.optimize = optimizer.apply_gradients(zip(self.actor_gradients, acto
    r_grad_params))
70          if self.actor_auxiliary_prediction:
71              self.optimize_prediction = optimizer.minimize(loss=self.auxil_loss,
72                                                            var_list=self.network_
    params)
73          commission_loss = self.auxiliary_commission* \
74              tf.reduce_mean(tf.reduce_sum(tf.square(self.scaled_out - self.portfo
    lio_inputs), axis=-1))
75          self.optimize_comm = optimizer.minimize(loss=commission_loss,
76                                                  var_list=self.network_params)
77
78          self.num_trainable_vars = len(self.network_params) + len(self.target_net
    work_params)
79
80      def create_actor_network(self):
81          raise NotImplementedError('Create actor should return (inputs, out, scaled_out)')
82
83      def train(self, inputs, a_gradient):
84          self.sess.run(self.optimize, feed_dict={
85              self.inputs: inputs,
86              self.action_gradient: a_gradient
87          })
88
89      def predict(self, inputs):
90          return self.sess.run(self.scaled_out, feed_dict={
91              self.inputs: inputs
92          })
93
94      def predict_target(self, inputs):
95          return self.sess.run(self.target_scaled_out, feed_dict={
96              self.target_inputs: inputs
97          })
98
99      def update_target_network(self):
100         self.sess.run(self.update_target_network_params)
101
102     def get_num_trainable_vars(self):
103         return self.num_trainable_vars
```

critic.py

```
"""
Modified from https://github.com/vermouth1992/drl-portfolio-management/blob/master/src/model/ddpg/critic.py
"""

import tensorflow as tf
import tflearn


class CriticNetwork(object):
    """
    Input to the network is the state and action, output is Q(s,a).
    The action must be obtained from the output of the Actor network.
    """

    def __init__(self, sess, state_dim, action_dim, learning_rate, tau, num_actor_vars):
        self.sess = sess
        assert isinstance(state_dim, list), 'state_dim must be a list.'
        self.s_dim = state_dim
        assert isinstance(action_dim, list), 'action_dim must be a list.'
        self.a_dim = action_dim
        self.learning_rate = learning_rate
        self.tau = tau

        # Create the critic network
        self.inputs, self.action, self.out, self.portfolio_inputs, self.auxil_loss, self.future_y_inputs \
            = self.create_critic_network(False)

        self.network_params = tf.trainable_variables()[num_actor_vars:]

        # Target Network
        self.target_inputs, self.target_action, self.target_out, self.target_portfolio_inputs, \
        self.target_auxil_loss, self.target_future_y_inputs = self.create_critic_network(True)

        self.target_network_params = tf.trainable_variables()[(len(self.network_params) + num_actor_vars):]

        # Op for periodically updating target network with online network
        # weights with regularization
        self.update_target_network_params = \
            [self.target_network_params[i].assign(tf.multiply(self.network_params[i], self.tau) \
                                                  + tf.multiply(self.target_network_params[i], 1. - self.tau))
             for i in range(len(self.target_network_params))]

        # Network target (y_i)
        self.predicted_q_value = tf.placeholder(tf.float32, [None, 1])

        # Define loss and optimization Op
        self.loss = tflearn.mean_square(self.predicted_q_value, self.out)
        self.loss += self.auxil_loss
        self.optimize = tf.train.AdamOptimizer(
            self.learning_rate).minimize(self.loss)

        # Get the gradient of the net w.r.t. the action.
        # For each action in the minibatch (i.e., for each x in xs),
        # this will sum up the gradients of each critic output in the minibatch
        # w.r.t. that action. Each output is independent of all
        # actions except for one.
```

```
57          self.action_grads = tf.gradients(self.out, self.action)
58
59      def create_critic_network(self):
60          raise NotImplementedError('Create critic should return (inputs, action, out)')
61
62      def train(self, inputs, action, predicted_q_value, future_y_inputs):
63          return self.sess.run([self.out, self.optimize], feed_dict={
64              self.inputs: inputs,
65              self.action: action,
66              self.predicted_q_value: predicted_q_value,
67              self.future_y_inputs: future_y_inputs
68          })
69
70      def predict(self, inputs, action):
71          return self.sess.run(self.out, feed_dict={
72              self.inputs: inputs,
73              self.action: action
74          })
75
76      def predict_target(self, inputs, action):
77          return self.sess.run(self.target_out, feed_dict={
78              self.target_inputs: inputs,
79              self.target_action: action
80          })
81
82      def action_gradients(self, inputs, actions):
83          return self.sess.run(self.action_grads, feed_dict={
84              self.inputs: inputs,
85              self.action: actions
86          })
87
88      def update_target_network(self):
89          self.sess.run(self.update_target_network_params)
```

ddpg.py


```
"""
Modified from https://github.com/vermouth1992/drl-portfolio-management/blob/master/src/model/ddpg/ddpg.py
"""
from __future__ import print_function

import matplotlib
matplotlib.use('Agg')

import os
import traceback
import json
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

from collections import deque
from copy import copy
from .replay_buffer import ReplayBuffer, ReplayBufferMultiple, ReplayBufferRollout
from ..base_model import BaseModel


def build_summaries():
    episode_reward = tf.Variable(0.)
    tf.summary.scalar("Reward", episode_reward)
    episode_ave_max_q = tf.Variable(0.)
    tf.summary.scalar("Qmax_Value", episode_ave_max_q)

    summary_vars = [episode_reward, episode_ave_max_q]
    summary_ops = tf.summary.merge_all()

    return summary_ops, summary_vars


class DDPG(BaseModel):
    def __init__(self, env, sess, actor, critic, actor_noise, obs_normalizer=None, action_processor=None,
                 gamma=0.5, training_episodes=600, max_rollout_steps=1000, buffer_size=100000, seed=1337, batch_size=64,
                 model_save_path='weights/ddpg/ddpg.ckpt', summary_path='results/ddpg/', infer_path='infer/',
                 infer_train_env=None, infer_test_env=None, learning_steps=1):
        np.random.seed(seed)
        if env:
            env.seed(seed)
        self.model_save_path = model_save_path
        self.summary_path = summary_path
        self.infer_path = infer_path
        self.sess = sess
        # if env is None, then DDPG just predicts
        self.env = env
        self.actor = actor
        self.critic = critic
        self.actor_noise = actor_noise
        self.obs_normalizer = obs_normalizer
        self.action_processor = action_processor
        self.gamma = gamma
        self.training_episodes = training_episodes
        self.max_rollout_steps = max_rollout_steps
        self.buffer_size = buffer_size
        self.seed = seed
        self.batch_size = batch_size
        self.infer_train_env = infer_train_env
```

```
60          self.infer_test_env = infer_test_env
61          self.learning_steps = learning_steps
62          self.start_episode = 0
63          self.summary_ops, self.summary_vars = build_summaries()
64
65      def clear_path(self, folder):
66          for file in os.listdir(folder):
67              file_path = os.path.join(folder, file)
68              try:
69                  if os.path.isfile(file_path):
70                      os.unlink(file_path)
71                  #elif os.path.isdir(file_path): shutil.rmtree(file_path)
72              except Exception as e:
73                  print(e)
74
75      def initialize(self, load_weights=True, verbose=True):
76          """ Load training history from path. To be add feature to just load weights, not training states
77
78      """
79
80          if (self.model_save_path is not None) and (not os.path.exists(self.model
    _save_path)):
81              os.makedirs(self.model_save_path, exist_ok=True)
82          if (self.summary_path is not None) and (not os.path.exists(self.summary_
    path)):
83              os.makedirs(self.summary_path, exist_ok=True)
84          if (self.infer_path is not None) and (not os.path.exists(os.path.join(se
    lf.infer_path, 'test/'))):
85              os.makedirs(os.path.join(self.infer_path, 'test/'), exist_ok=True)
86          if (self.infer_path is not None) and (not os.path.exists(os.path.join(se
    lf.infer_path, 'train/'))):
87              os.makedirs(os.path.join(self.infer_path, 'train/'), exist_ok=True)
88
89          if load_weights:
90              try:
91                  variables = tf.global_variables()
92                  param_dict = {}
93                  saver = tf.train.Saver()
94                  latest_checkpoint = tf.train.latest_checkpoint(self.model_save_p
    ath)
95                  print("LOADING FROM:", self.model_save_path)
96                  self.start_episode = int(latest_checkpoint.split('-')[1]) + 1
97                  saver.restore(self.sess, latest_checkpoint)
98                  for var in variables:
99                      var_name = var.name[:-2]
100                     if verbose:
101                         print('Loading {} from checkpoint. Name: {}'.format(var.name, var
    _name))
102                     param_dict[var_name] = var
103             except:
104                 traceback.print_exc()
105                 print('Build model from scratch')
106                 self.sess.run(tf.global_variables_initializer())
107         else:
108             print('Build model from scratch')
109             self.clear_path(self.model_save_path)
110             self.clear_path(self.summary_path)
111             self.clear_path(os.path.join(self.infer_path, 'test'))
112             self.clear_path(os.path.join(self.infer_path, 'train'))
113             self.sess.run(tf.global_variables_initializer())
114
115     def train(self, save_every_episode=1, verbose=True, debug=False):
116         """ Must already call intialize
```

```
117
118     Args:
119       save_every_episode:
120       print_every_step:
121       verbose:
122       debug:
123
124     Returns:
125
126     """
127         writer = tf.summary.FileWriter(self.summary_path, self.sess.graph)
128
129         self.actor.update_target_network()
130         self.critic.update_target_network()
131
132         np.random.seed(self.seed)
133         num_episode = self.training_episodes
134         batch_size = self.batch_size
135         gamma = self.gamma
136         self.buffer = ReplayBufferRollout(self.buffer_size)
137
138         # main training loop
139         for i in range(self.start_episode, num_episode):
140             if verbose and debug:
141                 print("Episode: " + str(i) + " Replay Buffer " + str(self.buffer.count(
    )))
142
143             episode_rollout = deque()
144
145             observation_1 = self.env.reset()
146             observation_1, weights_1 = observation_1[0]['obs'], observation_1[0]
    ['weights']
147
148             if self.obs_normalizer:
149                 observation_1 = self.obs_normalizer(observation_1)
150
151             episode_rollout.append([observation_1, weights_1])
152
153             for rollout_step in range(self.learning_steps - 1):
154                 obs, ws = episode_rollout[-1]
155                 action = self.actor.predict(inputs=np.expand_dims(obs, axis=0),
156                                             portfolio_inputs=np.expand_dims(ws,
    axis=0)).squeeze(
157                                             axis=0) + self.actor_noise()
158                 action = np.clip(action, 0, 1)
159                 if action.sum() == 0:
160                     action = np.ones(obs.shape[0])/obs.shape[0]
161                 action /= action.sum()
162                 new_obs, reward, done, info = self.env.step(action)
163                 new_obs, new_ws = new_obs['obs'], new_obs['weights']
164
165                 if self.obs_normalizer:
166                     new_obs = self.obs_normalizer(new_obs)
167                 episode_rollout.append(action)
168                 episode_rollout.append(reward)
169                 episode_rollout.append(done)
170                 episode_rollout.append(info['next_y1'])
171                 episode_rollout.append([new_obs, new_ws])
172
173             ep_reward = 0
174             ep_ave_max_q = 0
175             ep_ave_min_q = 0
176             # keeps sampling until done
```

```
177             for j in range(self.max_rollout_steps):
178                 #print(j)
179                 action = self.actor.predict(inputs=np.expand_dims(episode_rollou
    t[-1][0], axis=0),
180                                             portfolio_inputs=np.expand_dims(epis
    ode_rollout[-1][1], axis=0)).squeeze(
181                     axis=0) + self.actor_noise()
182 
183                 if self.action_processor:
184                     action = self.action_processor(action)
185                 else:
186                     action = action
187 
188                 action = np.clip(action, 0, 1)
189                 if action.sum() == 0:
190                     action = np.ones(episode_rollout[-1][0].shape[0])/episode_ro
    llout[-1][0].shape[0]
191                 action /= action.sum()
192 
193                 obs, reward, done, info = self.env.step(action)
194                 obs, ws = obs['obs'], obs['weights']
195 
196                 if self.obs_normalizer:
197                     obs = self.obs_normalizer(obs)
198 
199                 episode_rollout.append(action)
200                 episode_rollout.append(reward)
201                 episode_rollout.append(done)
202                 episode_rollout.append(info['next_y1'])
203                 episode_rollout.append([obs, ws])
204 
205                 # add to buffer
206                 self.buffer.add(copy(episode_rollout))
207 
208                 if self.buffer.size() >= batch_size:
209                     # batch update
210 
211                     s1_batch, s1w_batch, a1_batch, s1y_batch, rs_batch, \
212                         t_batch, sf_batch, sfw_batch = self.buffer.sample_batch(
    batch_size)
213 
214                     # Calculate targets
215                     target_q = self.critic.predict_target(inputs=sf_batch,
216                                                           action=self.actor.pred
    ict_target(inputs=sf_batch,
217 
            portfolio_inputs=sfw_batch),
218                                                           portfolio_inputs=sfw_b
    atch)
219 
220                     y_i = []
221                     for k in range(batch_size):
222                         total_r = 0
223                         for r_batch in reversed(rs_batch):
224                             total_r *= gamma
225                             total_r += r_batch[k]
226                         if t_batch[k]:
227                             y_i.append(total_r)
228                         else:
229                             y_i.append(total_r + (gamma**len(rs_batch))*target_q
    [k])
230 
231                     # Update the critic given the targets
```

```
                    predicted_q_value, _ = self.critic.train(inputs=s1_batch,
                                                             action=a1_batch,
                                                             predicted_q_value=np.reshape(y_i, (batch_size, 1)),
                                                             portfolio_inputs=s1w_batch,
                                                             future_y_inputs=s1y_batch)

                    ep_ave_max_q += np.amax(predicted_q_value)
                    ep_ave_min_q += np.amin(predicted_q_value)

                    # Update the actor policy using the sampled gradient
                    a_outs = self.actor.predict(inputs=s1_batch,
                                                portfolio_inputs=s1w_batch)
                    grads = self.critic.action_gradients(inputs=s1_batch,
                                                         actions=a_outs,
                                                         portfolio_inputs=s1w_batch)
                    self.actor.train(inputs=s1_batch,
                                     a_gradient=grads[0],
                                     portfolio_inputs=s1w_batch,
                                     future_y_inputs=s1y_batch)

                    # Update target networks
                    self.actor.update_target_network()
                    self.critic.update_target_network()

                ep_reward += reward
                [episode_rollout.popleft() for _ in range(5)]

                if done or j == self.max_rollout_steps - 1:
                    summary_str = self.sess.run(self.summary_ops, feed_dict={
                        self.summary_vars[0]: ep_reward,
                        self.summary_vars[1]: ep_ave_max_q / float(j)
                    })

                    writer.add_summary(summary_str, i)
                    writer.flush()

                    if (i % 10) == 0:
                        print("INFERRING")
                        self.infer(i, True)
                        self.infer(i, False)

                    if ((i+1) % 50) == 0:
                        print("SAVING")
                        self.save_model(i, 7, verbose=True)

                    print('Episode: {:d}, Reward: {:.2f}, Qmax: {:.4f}, Qmin{:.4f}'.format(i,
                        ep_reward, (ep_ave_max_q / float(j)), (ep_ave_min_q / float(j))))
                    break

        self.save_model(i, 7, verbose=True)
        print('Finish.')

    def predict(self, observation):
        """ predict the next action using actor model, only used in deploy.
        Can be used in multiple environments.

    Args:
        observation: (batch_size, num_stocks + 1, window_length)
```

```
290
291       Returns: action array with shape (batch_size, num_stocks + 1)
292
293       """
294           if self.obs_normalizer:
295               observation = self.obs_normalizer(observation)
296           action = self.actor.predict(observation)
297           if self.action_processor:
298               action = self.action_processor(action)
299           return action
300
301       def predict_single(self, observation):
302           """ Predict the action of a single observation
303
304       Args:
305         observation: (num_stocks + 1, window_length)
306
307       Returns: a single action array with shape (num_stocks + 1,)
308
309       """
310           observation, weights = observation['obs'], observation['weights']
311
312           if self.obs_normalizer:
313               observation = self.obs_normalizer(observation)
314           action = self.actor.predict(inputs=np.expand_dims(observation, axis=0),
315                                       portfolio_inputs=np.expand_dims(weights, axi
    s=0)).squeeze(axis=0)
316           if self.action_processor:
317               action = self.action_processor(action)
318           return action
319
320       def save_model(self, episode, max_to_keep=5, verbose=False):
321           if not os.path.exists(self.model_save_path):
322               os.makedirs(self.model_save_path, exist_ok=True)
323
324           saver = tf.train.Saver(max_to_keep=max_to_keep)
325           model_path = saver.save(self.sess, os.path.join(self.model_save_path, "c
    heckpoint.ckpt"),
326                                   global_step=episode)
327           print("Model saved in %s" % model_path)
328
329       def infer(self, episode, train):
330           """ Must already call intialize
331       """
332           if not train:
333               env = self.infer_test_env
334           else:
335               env = self.infer_train_env
336
337           episode_rollout = deque()
338
339           observation_1 = env.reset()
340           observation_1, weights_1 = observation_1[0]['obs'], observation_1[0]['wei
    ghts']
341
342           if self.obs_normalizer:
343               observation_1 = self.obs_normalizer(observation_1)
344
345           episode_rollout.append([observation_1, weights_1])
346
347           for rollout_step in range(self.learning_steps - 1):
348               obs, ws = episode_rollout[-1]
349               action = self.actor.predict(inputs=np.expand_dims(obs, axis=0),
```

```
                                            portfolio_inputs=np.expand_dims(ws, axis
=0)).squeeze(
                                            axis=0)
            action = np.clip(action, 0, 1)
            if action.sum() == 0:
                action = np.ones(obs.shape[0])/obs.shape[0]
            action /= action.sum()
            new_obs, reward, done, _ = env.step(action)
            new_obs, new_ws = new_obs['obs'], new_obs['weights']

            if self.obs_normalizer:
                new_obs = self.obs_normalizer(new_obs)
            episode_rollout.append(action)
            episode_rollout.append(reward)
            episode_rollout.append(done)
            episode_rollout.append([new_obs, new_ws])

        for j in range(env.steps-self.learning_steps):
            action = self.actor.predict(inputs=np.expand_dims(episode_rollout[-1
][0], axis=0),
                                            portfolio_inputs=np.expand_dims(episode_
rollout[-1][1], axis=0)).squeeze(
                                            axis=0)

            if self.action_processor:
                action = self.action_processor(action)
            else:
                action = action

            action = np.clip(action, 0, 1)
            if action.sum() == 0:
                action = np.ones(episode_rollout[-1][0].shape[0])/episode_rollou
t[-1][0].shape[0]
            action /= action.sum()

            obs, reward, done, _ = env.step(action)
            obs, ws = obs['obs'], obs['weights']

            if self.obs_normalizer:
                obs = self.obs_normalizer(obs)

            episode_rollout.append(action)
            episode_rollout.append(reward)
            episode_rollout.append(done)
            episode_rollout.append([obs, ws])

            [episode_rollout.popleft() for _ in range(4)]

            if done or j == env.steps-self.learning_steps-1:
                label = 'train' if train else 'test'
                env.render()
                plt.savefig(os.path.join(self.infer_path, label + '/', str(episo
de)+".png"))
                plt.close()
                break
```