Schulich School of Business

FINE 6901 WF19: Individual Study

Deep Reinforcement Learning Architecture for Portfolio Optimization

Nov 2, 2019
Daniel Fudge
215868904

Executive Summary

This report in concert with the Udacity Nanodegree in Deep Reinforcement Learning concludes the 2nd term Independent Study into Machine Learning applications to portfolio optimization. The 1st term was a general investigation into the "Application of Machine Learning to Portfolio Optimization" that proposed the development of a Reinforcement Learning framework to better understand its possible applications.

In response to that proposal, this report formally defines the problem statement, summarizes the related concepts and proposes an architecture and development environment. For this implementation it also identifies research that must be executed prior to its implementation and future work to extend the implementation.

Table of Contents

Background

This report is a component of a larger 3-term Independent Study with Professor Yelena as part of a concurrent Master of Business Administration (MBA) and Diploma in Financial Engineering from the Schulich School of Business.

The first term was a general investigation into applications of machine learning to portfolio optimization. Here we reviewed the different aspects of machine learning and their possible applications to Portfolio Optimization. During this investigation we highlighted Deep Reinforcement Learning (DRL) as an especially promising area to research and proposed the development of a DRL framework to better understand its applications. The report capturing this research may be found at the following link.

https://github.com/daniel-fudge/DRL-Portfolio-Optimization/blob/master/docs/report1.pdf

The second term, which this report summarizes, pursues the future work identified in the first term. To get a better understanding of DRL, the Udacity DRL Nanodegree was completed. A description of this nanodegree can be found at the link below.

https://www.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893

This Nanodegree involved developing DRL networks to solve 3 different Unity ML-Agents environments. The solutions to these environments can be found in the following GitHub repositories.

1. Banana Collector - https://github.com/daniel-fudge/banana_hunter
2. Reacher - https://github.com/daniel-fudge/reinforcement-learning-reacher
3. Tennis - https://github.com/daniel-fudge/reinforcement-learning-tennis

With the Udacity Nanodegree complete, the next step is formally defining the problem statement and associated solution architecture, as captured in this report. With the problem statement and architecture defined, the task of the third term will be to implement the architecture. Note that the intent of this implementation is not to advance the state-of-the-art in DRL or its application to portfolio optimization. Instead it is to generate a functioning DRL platform for portfolio optimization. From this conventional implementation, we can experiment with more advanced techniques.

Reinforcement Learning (RL) Review

At a high level, RL is the process of learning the optimal strategy (***Policy***) that defines ***Actions*** that an ***Agent*** should take given the current ***State***, ***Reward*** function, and future reward ***Discount Factor***. The figure below illustrates the classic RL feedback loop used to determine the optimal policy. Here, the environment is modeled as a function that predicts the next reward and state given the current state and action.

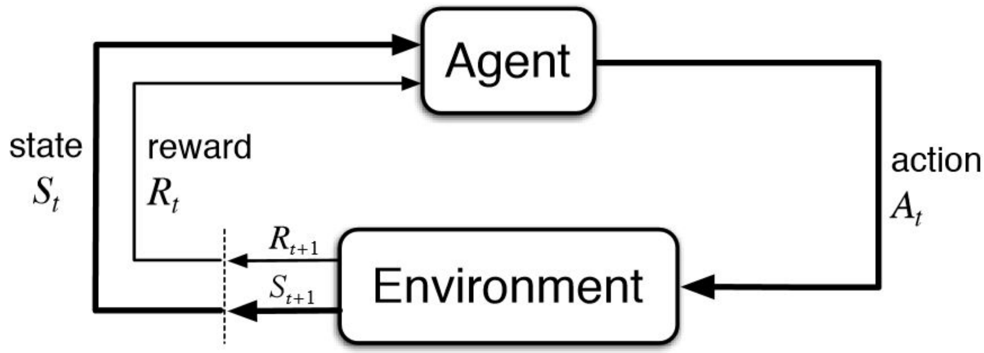


Figure 1. Reinforcement Learning feedback loop.
Image source: https://i.stack.imgur.com/eoeSq.png

Definitions

Before we begin the discussion we need to make the following definitions.[1]

1. Agent: The algorithm or person that takes the given action.
2. Environment: The world in which the agents exists. It converts the agent's current state and actions into the associated rewards and next state.
3. Action (***a***): All the possible moves that the agent can take.
4. State (***s***): Current situation returned by the environment.
5. Reward (***R***): An immediate return sent back from the environment in response to the last action.
6. Policy (π): The strategy that the agent employs to determine the next action based on the current state.
7. Value (***V***): The expected long-term return with discount, as opposed to the short-term reward ***R***. ***V***π(***s***) is defined as the expected long-term return of the current state ***s*** under policy π.
8. Q-value or action-value (***Q***): Similar to ***V***, except that it takes an extra parameter, the current action ***a***. ***Q***π(***s***, ***a***) refers to the long-term return of the current state ***s***, taking action ***a*** under policy π.
9. Discount Factor (γ): The discount factor applied to future rewards.

[1]Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. 2nd edition, Cambridge, Massachusetts: *The MIT Press*, 2018.

Adding "Deep" to RL

Reinforcement learning can be implemented traditionally as an extension of dynamic programming, which requires a model of the environment. For instance, a classic model-based approach for option pricing uses a stochastic differential equation such as the Black-Scholes-Merton (BSM) model of the environment.[2] It then uses standard dynamic programming techniques and the Bellman optimality equation as an action-value function to identify the optimal policy.

In a model-free approach we do not assume a model of the environment, instead we deploy one or more deep neural networks to learn from the environment and build the optimal policy directly. As shown below a neural network becomes deep as the number of hidden layers increase, which allows more complex relationships to be modeled. In later sections we will discuss special networks such as CNN, RNN and LSTM that can be designed to extract spatial and sequential structure from the input space.

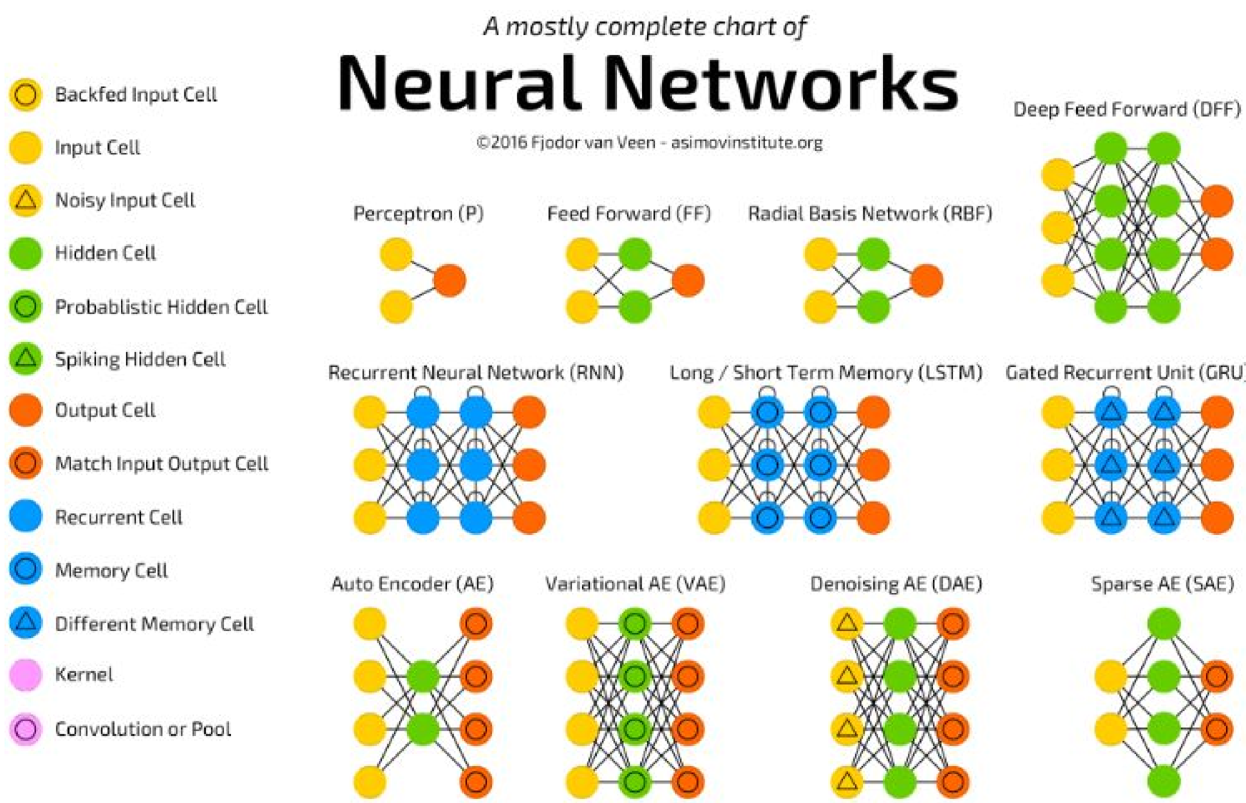


Figure 2. Neural network architectures.[3]
Image source: https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/neural_networks.html

---

[2] Igor Halperin. "QLBS: Q-Learner in the Black-Scholes(-Merton) Worlds." *SSRN Electronic Journal*, December 2017, DOI: 10.2139/ssrn.3087076, p2.

[3] Leonardo Araujo dos Santos. "Artificial Intelligence." *GitBook*, accessed October 6, 2019. https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/

Problem Statement

The problem statement is the state space, rewards, action space and objective.

State Space

Since we are performing portfolio optimization, the first obvious state selection will be the prices of the universe of assets we may place in our portfolio. We assume our trades have a minimal market impact and can be executed rapidly at the given price, therefore these assets must be highly liquid and have a large market capitalization in comparison to our portfolio. Therefore we will randomly choose *n* assets from the S&P 500 that have data for the last 12 years; 10 years for training and 2 years for testing. Considering the S&P 500 contains the 500 largest US companies on the NYSE or NASDAQ with a total capitalization of nearly $26 trillion (9/30/2019), the assets have both the liquidity and capitalization we desire.[4] Additionally, there is a large body of research related to the S&P 500 and Reinforcement Learning as indicated by the 3,200 articles listed by the Google Scholar search for "S&P 500 reinforcement learning".

The next decision is what time scale we wish to model. On one extreme is high frequency trading, which operates on the sub-second time scale. A major advantage of this scale is the quantity of data available to train our model. More data allows us to train more complex networks that can extract extremely complex interactions. However, to be successful at this time scale we require the ability to access and react to the data as fast as any other actor. A longer time scale is monthly data, which gives us plenty of time to react, but doesn't provide enough data to model complex interactions before the environment shifts and reduces the accuracy of the data. For instance, in 5 years there are only 60 monthly data points, which is completely insufficient to build an accurate deep neural net. Therefore we will select daily prices as a compromise.

To capture some of the volatility in the daily data, we will also select the closing, high and low prices ($\mathbf{P}, \mathbf{P}^{\mathbf{H}}, \mathbf{P}^{\mathbf{L}}$). We will also assume the markets violate the weak form efficiency of the efficient market hypothesis and react to momentum, therefore we will include k days of prices.[5] To add a signal for the market, we will add the S&P 500 index to the n assets. This gives a [3, n+1, k] price tensor $\mathbf{Y_t}$.

$\mathbf{Y_t} = [\mathbf{P}, \mathbf{P^H}, \mathbf{P^L}]$ with $p_{i,j}$ for asset $i \in [0, n]$ and time embedding $j \in [t-k, t]$

[4] Siblis Research. "S&P 500 Historical Total Market Cap & Float Adjusted Cap." September 30, 2019.

[5] Stephen Ross et al.. "The Efficient Markets Hypothesis" in *Fundamentals of Corporate Finance*, 9th edition, McGraw-Hill Education, 2019, pp 448-449.

Note that this state tensor could easily be expanded to include a wide assortment of signals such as GDP, inflation rates, interest rates, company financial ratios or even twitter sentiment. This purpose of this study is to develop the architecture, the signal selection is a never ending area of research.

We also need to add the portfolio weighting $\mathbf{w}$ as defined below to define the full state space $\mathbf{S_t}$. Note asset 0 is set to cash for simplicity.

$$w_{i,t} \quad \textit{for asset } i \in [0,\ n],\ \sum_{i=0}^{n} w_{i,t} = 1 \quad \textit{for all } t \text{ and } \mathbf{w_o} = [1, 0, \ldots 0]^{\top}$$

$$\mathbf{S_t} \doteq [\mathbf{Y_t}, \mathbf{w_t}]$$

There is an additional practical consideration of allowing the weights to be negative to indicate short selling. There is nothing to mathematically prevent us from allowing short selling and the market for the S&P 500 assets will allow it. This may create very risky portfolios but the network should learn this risk trade. Both allowing and preventing shorting are included in the literature but I haven't read a discussion on the impact of shorting on the learning algorithm. Training the network with and without the non-negativity constraint would be interesting. If the shorter appears to degrade performance, intermediate limitations could also be placed on the magnitude of the individual short positions; $w_i > -a$.

Reward Function

The reward function r_t has three components. The first component is the change in portfolio value from t to $t+1$, which we will denote $\Delta\rho_t$. To the n S&P assets we will also carry cash to make a portfolio of $n+1$ assets. To calculate the change in portfolio value we must consider the transaction cost of changing the portfolio weighting $\mathbf{w}$ as defined below. Note c is a fee for selling or purchasing assets. A more accurate representation of these fees is left for future work.

Due to the changes in asset prices between t-1 and t, we must define the weighting $\mathbf{w'_t}$ before the trades at time t as follows. This uses the gross return ratio $u_{i,t}$ with $u_{0,t}$ = 1, since asset i is cash. Note $\odot$ and $\oslash$ are element-wise multiplication and division respectively.

$$\mathbf{u_t} \doteq \mathbf{P_t} \oslash \mathbf{P_{t-1}} = [1, p_{1,t}/p_{1,t-1}, \ldots, p_{n,t}/p_{n,t-1}]^{\top}$$

$$\mathbf{w'_t} = (\mathbf{u_t} \odot \mathbf{w_{t-1}})/(\mathbf{u_t} \cdot \mathbf{w_{t-1}})$$

$$\mathbf{\Delta w_t} = \mathbf{w_t} - \mathbf{w'_t}$$

$$\overline{c_t} \doteq 1 - c\sum_{i=0}^{n} |\Delta w_{i,t}|, \text{ where c is a constant} \geq 0$$

$$\Delta\rho_t \doteq \ln(\overline{c_t}\mathbf{u_t} \cdot \mathbf{w_{t-1}})$$

The second reward component is added to penalize for risk as indicated by large drops in the portfolio value over a given number of time steps m, denoted as d_t. Setting m > 1, prevents the penalization of short term volatility as long as the drops are recovered in the next m time steps. Note this also requires the recording of the portfolio value ρ_t for the previous m time steps, where ρ_0 is set to 1 for simplicity.

$$\rho_t \doteq \rho_{t-1}\overline{c_t}\mathbf{u_t} \cdot \mathbf{w_{t-1}}$$

$$d_t \doteq -ln(min(1, \rho_t/\rho_{t-m}))^2$$

The third component is simply a binary gain g_t applied to the reward that is set to zero if the portfolio value hits zero, indicating a default. In practice, the code will simply end the episode and set the remaining rewards to zero. Therefore the reward is the sum below.

$$r_t \doteq \Delta\rho_t + d_t = ln(\overline{c_t}\mathbf{u_t} \cdot \mathbf{w_{t-1}}) - ln(min(1, \rho_t/\rho_{t-m}))^2 \qquad [1]$$

Action Space

From the above reward function it can be seen that the action space is simply $\mathbf{\Delta w_t}$.

Objective

To define the objective we also need a discount factor γ, an investment time horizon T and portfolio value ϱ_t. Therefore the objective is to maximize the discounted expected portfolio value at time T as defined below.

$$MAX : \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r_t\right] \qquad [2]$$

Methodology Review

Now that the problem statement is defined our task is to build a network that learns the optimal policy (π) that select the actions (**a**) for a given state (**s**) to maximize the objective. There are a wide variety of network architectures to solve this problem. The list below is a short summary of the popular methods with features that will be incorporated into the chosen network.

Deep Q-Learning (DQN)[6]

A deep neural network is generated that learns the **Q**-value of all possible actions. Therefore the optimal policy can simply take the greedy action that maximizes the **Q**-value for the given state. Often an ϵ-greedy action is taken where there is a probability ϵ that a random action will be taken. This allows for learning and exploration and the value of ϵ decays over time as the network becomes smarter.

Double DQN[7]

Google's DeepMind developed Double DQN to overcome DQN's overestimation of the **Q**-values by creating two identical networks; one for action selection (local) and one for evaluation (target). The local network that selects the actions is constantly learning from experience and uses the difference form the more stable target as a loss function. The target is then updated periodically from the local network with an update factor τ; target = τ*local + (1-τ)*target. A common value of τ is 0.001 but it is often tuned for a specific application. This type of update is referred to as a ***soft update***.

Prioritized Experience Replay (PER)[8]

Google's DeepMind also improved on DQN by prioritizing which experiences are used to train the model. An experience is a $\langle \mathbf{s}_t, \mathbf{a}_t, \mathbf{r}_{t+1}, \mathbf{s}_{t+1} \rangle$ tuple that records the current state and action and the resulting reward and state. In classical reinforcement learning, the network learns from the experience as it happens and then the experience is forgotten. Unfortunately, correlation between subsequent experiences impairs gradient-based algorithms used to train the network. Additionally possibly rare and valuable experiences are forgotten. To overcome these issues, a memory buffer is used to store the experiences, which can then be randomly sampled to break the temporal correlations and boost learning through repetition. Google's PER takes this a step further by extending the tuple to include a priority term that captures the value of the experience and then weighting the memory sampling by this priority value.

An excellent comparison of seven DQN variants and their combination (Rainbow) on 57 Atari games is shown below.

---

[6] Volodymyr Mnih, et al.. "Human level control through deep reinforcement learning." *Nature*, 518(7540): 529–533, 2015.

[7] Hado van Hasselt, et al. "Deep Reinforcement Learning with Double Q-Learning." *Google DeepMind*, 13th Conference on Artificial Intelligence (AAAI-16), 2016.

[8] Tom Schaul, et al. "Prioritized Experience Replay." *Google DeepMind*, ICLR 2016.

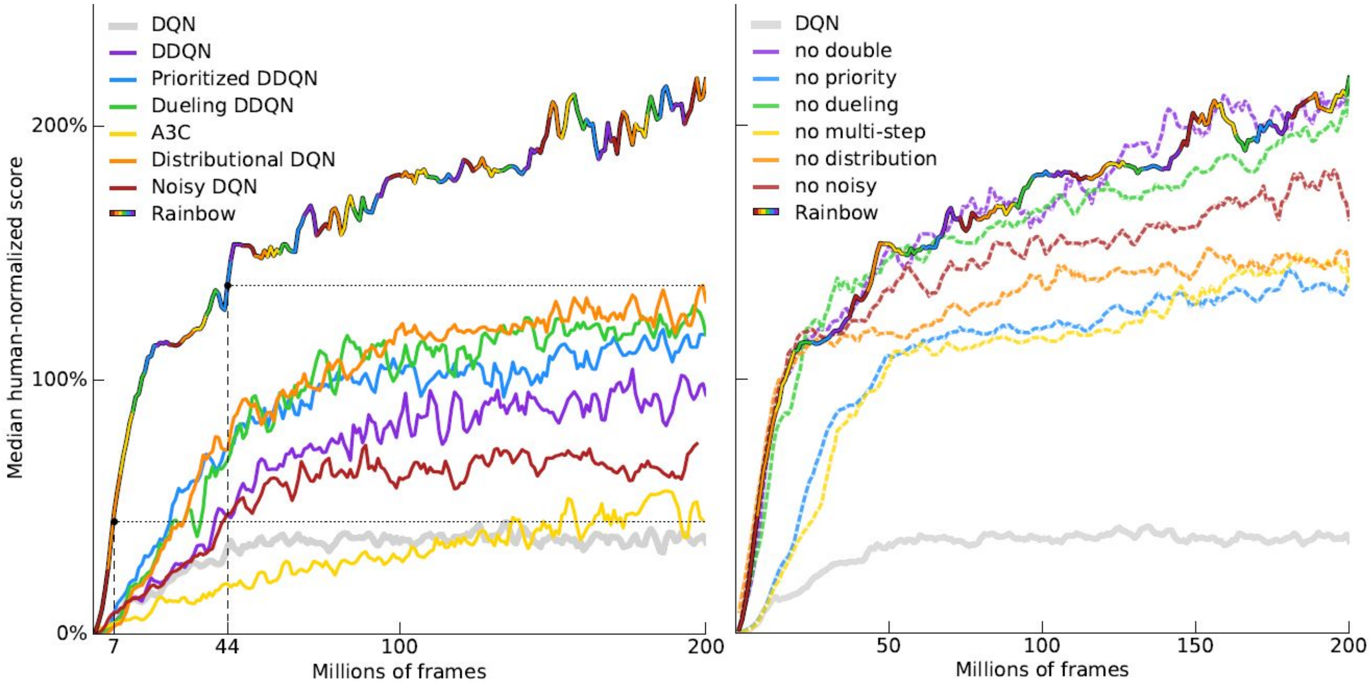


Figure 3. Google DeepMind's DQN variant comparison over 57 Atari games.[9]
Image source: https://arxiv.org/abs/1710.02298

The image on the right of the previous figure is especially telling. Removing PER (blue 'no priority') shows a considerable drop in performance, illustrating its power. Note that this performance difference may not be replicated in all applications but it is still suggestive.

Policy-Based Methods

Up to this point we have been discussing value-based methods that depend on creating a network to estimate the **Q**-value of the state-action pairs and then select the optimal action to maximize the **Q**-Value. The following policy-based algorithms skip the estimation of the **Q**-Value and directly learn the optimal policy (actions). A value-based method also requires discrete actions to perform the max value operation. In our application the action space Δw_t is continuous. We could discretize the action space but policy-based methods give us the ability to directly predict the optimal Δw_t in a continuous space.

The policy-based methods often incorporate a gradient-ascent algorithm to find the network weights that maximize the objective. This ranges from the classic

[9] Matteo Hessel, at al.. "Rainbow: Combining Improvements in Deep Reinforcement Learning." *Google DeepMind*, Association for the Advancement of Artificial Intelligence (AAAI), 2018.

gradient-based REINFORCE[10] method to Trust Region Policy Optimization (TRPO)[11] and more advanced Proximal Policy Optimization (PPO)[12], which overcomes some of the complexity of implementing other policy-based methods.

Deterministic Policy Gradient (DPG)[13]

Policy gradient algorithms are often stochastic with a parametric probability policy distribution $\pi_\theta(\mathbf{a}|\mathbf{s}) = \mathbb{P}[\mathbf{a}|\mathbf{s};\theta]$, which provides the probability of selecting action ($\mathbf{a}$) for a given state ($\mathbf{s}$) according to the parameter vector $\boldsymbol{\theta}$. In a deterministic policy gradient algorithm we train a network to directly determine $\mathbf{a} = \mu_\theta(\mathbf{s})$. Stochastic methods must integrate over both state and action spaces, whereas deterministic methods only integrate over the state space. This reduces the data requirement for training a deterministic method especially in an action space with high dimensionality.

---

[10] Ronald J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning." *Reinforcement Learning*. Springer, Boston, MA, 1992.5-32.

[11] John Schulman, et al. “Trust Region Policy Optimization.” *University of California*, 31st ICM L, 2015.

[12] John Schulman, at al. “Proximal Policy Optimization Algorithms.” *OpenAI*, August 28, 2017.

[13] David Silver, et al. “Deterministic Policy Gradient Algorithms.” *31st ICML*, Beijing, China, 2014.

Actor-Critic Methods[14]

Actor-Critic methods combine Value-based and Policy-based methods by creating two separate neural networks; the actor and critic. The actor is like a policy-based network, which selects which actions to take. The critic is similar to the value-based network and estimates the value of the actions chosen by the actor and generates the loss used to train the actor.

Deep Deterministic Policy Gradient (DDPG)

The authors of DDPG classified it as a special form of Actor-Critic method.[15] However Miguel Morales argued in the Udacity DRL Nanodegree that DDPG is best classified as a DQN method for continuous action spaces with normalized advantage functions.[16] Regardless of the classification, it is a relatively simple architecture to implement and is very effective in a wide range of applications including portfolio management.[17,18] To prepare for the application to portfolio management, a DDPG architecture was created and successfully solved 3 unity ML-Agent environments to illustrate its flexibility and effectiveness.[19]

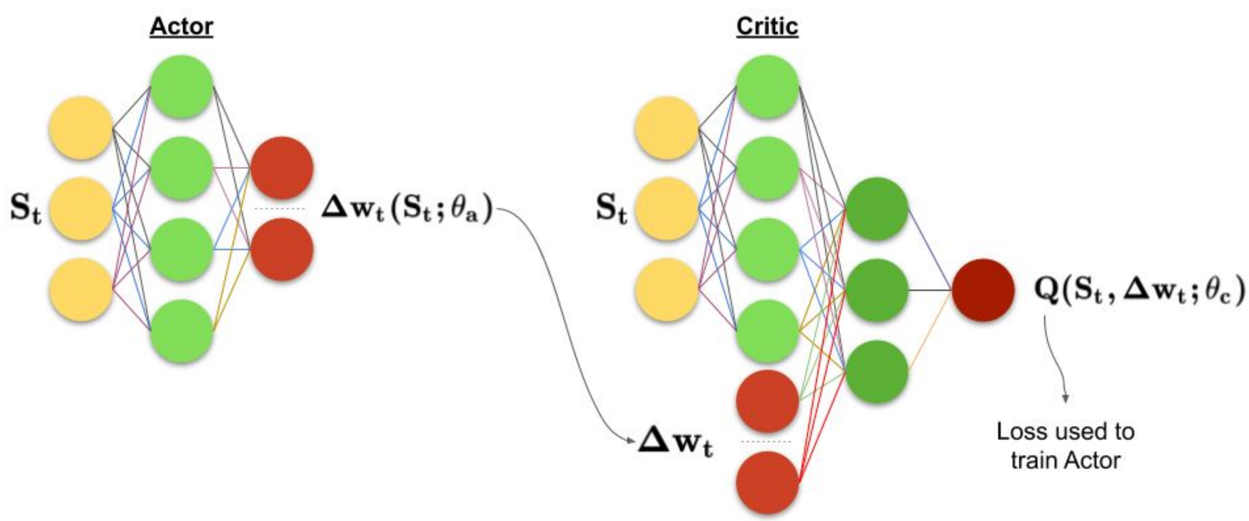


Figure 4. Generic DDPG network diagram.

The figure above illustrates the basic concept of the DDPG network as applied to our problem statement. The actor selects what action Δw_t to take based on the given

[14] Vijay R. Konda and John N. Tsitsiklis. “Actor-Critic Algorithms.” *MIT*, Cambridge, 2000.
[15] Timothy Lillicrap, et al. “Continuous Control With Deep Reinforcement Learning.” *ICLR* 2016.
[16] Shixiang Gu, et al. “Continuous Deep Q-Learning with Model-based Acceleration.” *Google*, 2016.
[17] Pengqian Yu, et al. “Model-based Deep RL for Financial Portfolio Optimization.” *ICML*, 2019.
[18] Shashank Hegde, et al. "Risk aware portfolio construction using DDPG," *IEEE SSCI*, 2018
[19] Daniel Fudge. “DRL-Portfolio-Optimization.” *GitHub Project*, 2019.

state $\mathbf{S}_t$ and the learned actor network weights $\theta_{\mathbf{a}}$. The critic then determines the $\mathbf{Q}$-value based on the state $\mathbf{S}_t$, the action selected be the actor $\Delta\mathbf{w}_t$ and the learned critic network weights $\theta_{\mathbf{c}}$. This $\mathbf{Q}$-value is then used to train the actor. Note the simple single layer fully connected network shown in the image does not reflect the actual network.

Ornstein-Uhlenbeck Noise Process[20]

All RL algorithms must balance the need to explore and exploit the action-state space to avoid getting caught in a local minimum while still learning the optimal policy as quickly as possible. A standard approach is to randomly perturb the predicted optimal action by applying noise to the selected actions. Another approach that appears very promising is to add the noise directly to the actor network weights $\theta_{\mathbf{a}}$.[21,22] A common noise process used in both cases is the Ornstein-Uhlenbeck (OU) noise process, which is a Gaussian process, a Markov process and is temporally homogeneous.[23]

Recurrent Neural Network (RNN)

Recurrent neural networks were developed to exploit temporal dependencies in sequence data such our price $p_{i,j}$ for asset $i \in [0, n]$ over the time embedding $j \in [t-k, t]$ as shown in the figure below.

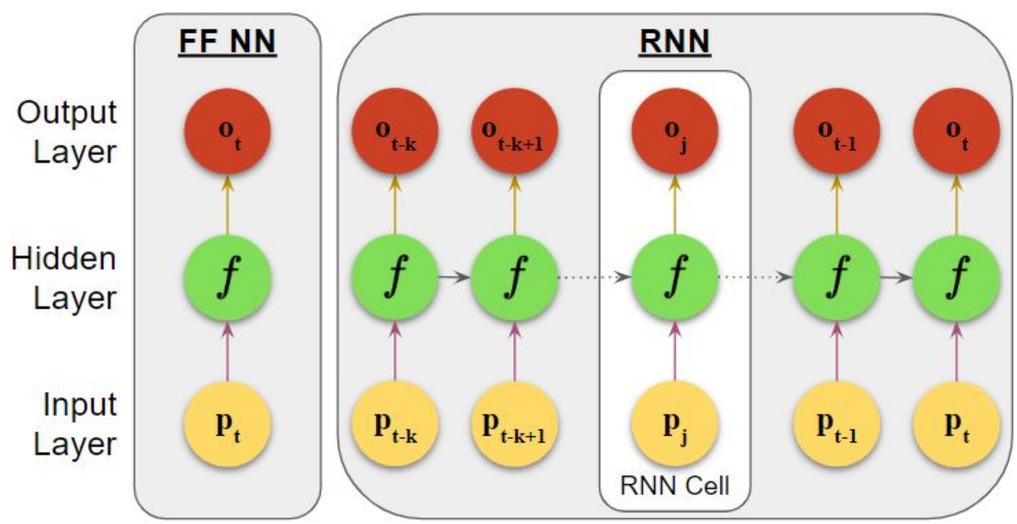


Figure 5. Feed-Forward (FF) vs. Recurrent Neural Network (RNN).

---

[20] George Uhlenbeck and Leonard Ornstein. "On the Theory of the Brownian Motion." *Physical review*, 1930.
[21] Matthias Plappert,et al. "Parameter Space Noise for Exploration." *OpenAI*, ICLR, 2016.
[22] Matthias Plappert,et al. "Parameter Space Noise for Exploration." *OpenAI*, Blog, 2017.
[23] J.T. Doob. "The Brownian Movement and Stochastic Equations." *Annals of Mathematics*, 1942.

The basic concept of a RNN is the output of the hidden layer is a function of not only the input layer, such as in a standard feed-forward network, but also of the hidden layer of the previous time step. The final cell may generate the desired output or the full RNN output layer may be passed to additional RNN layers to capture higher levels of complexity. Note for simplicity the above figure illustrates a 1D input (single asset). Much of the RNN literature also only addresses 1D input such as text or a single asset however this concept can be extended to multiple dimensions.[24]

Long Short-Term Memory (LSTM)[25]

A major short-coming of RNNs is the vanishing gradients experienced during the back-propagation through time while training the network. Effectively the impact of past events decay exponentially with time so important events are quickly forgotten. LSTM was introduced to overcome this issue and also add the ability to forget the past, which is important if an event occurs that makes previous information irrelevant.[26]

An excellent description of LSTM and RNN can be found in colah's blog, which I will condense and adapt for our application.[27] A LSTM cell which feeds information from one time step to the next like the RNN has three new features that generate memory that is both flexible and stable. The first is a cell state ($\mathbf{C}_j$) that represents the memory. The second is a sigmoid ($\boldsymbol{\sigma}$) function shown below, which being [0, 1], acts as a gate when multiplied to a signal. The last is a ***tanh*** function, which being [-1, 1], prevents a signal from exploding in either the positive or negative direction.

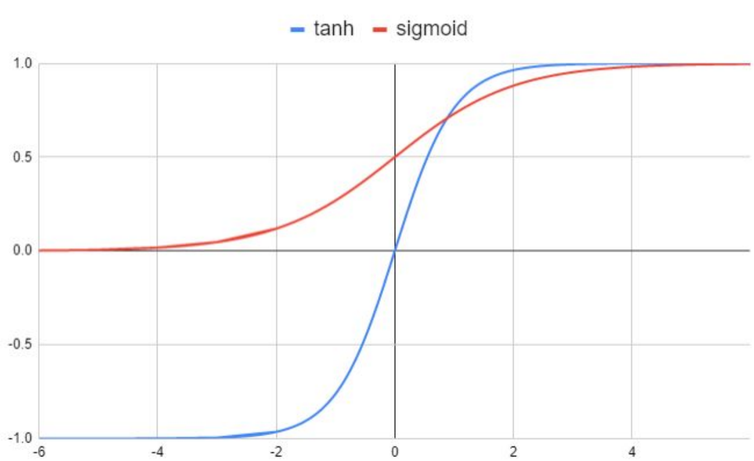


Figure 6. Sigmoid and tanh functions.

---

[24] Alex Graves, et al. "Multi-Dimensional Recurrent Neural Networks." *IDSIA*, 2013.
[25] Sepp Hochreiter and Jurgen Schmidhuber "Long short-term memory." *Neural computation*, 1997.
[26] Klaus Greff, et al. "LSTM: A Search Space Odyssey." *IEEE TNNLS*, Vol. 28, No. 10, pp. 2222-2232, 2017.
[27] Christopher Olah. "Understanding LSTM Networks." *colah's blog*, August 2015.

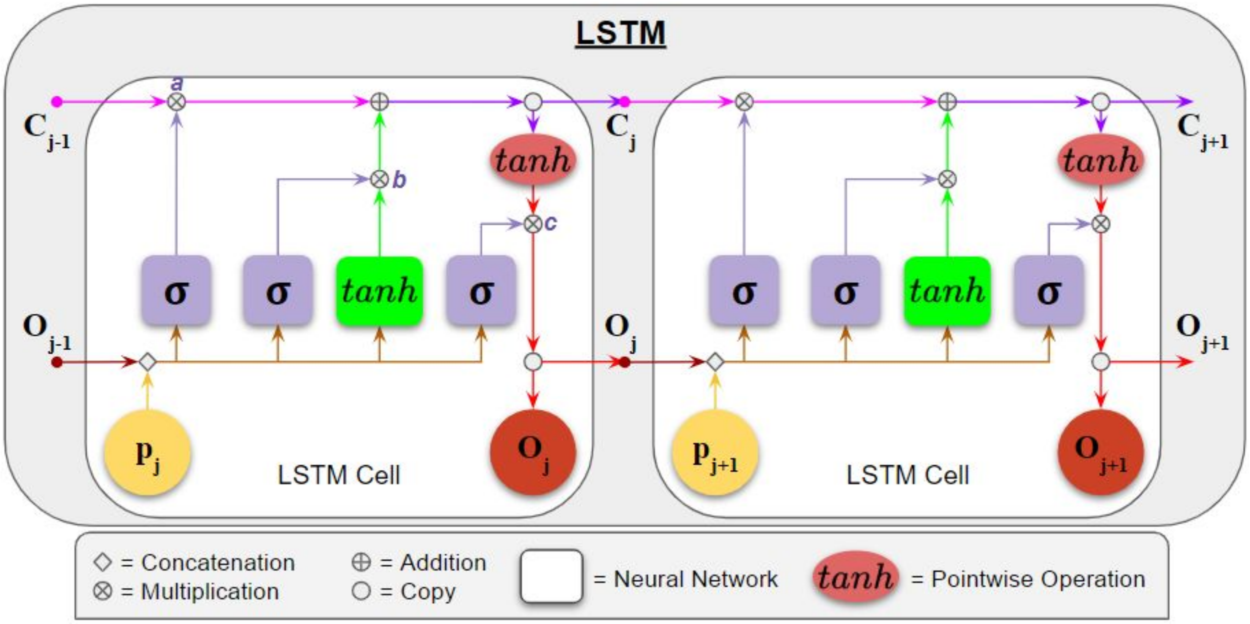


Figure 7. LSTM Network Architecture.

It is worth taking the time to study the above figure and compare it to the previous RNN figure. Both cells pass the output $\mathbf{O}_j$ from one time step to the next, however the LSTM also passes a hidden cell state $\mathbf{C}_j$. In addition, if the sigmoid feeding into the "***a***" multiplication is zero, the cell state is "forgotten". If the sigmoid feeding into the "***b***" multiplication is zero, the price signal $\mathbf{p}_j$ is ignored and the previous cell state is passed to the next cell. And finally if the sigmoid feeding into the "***c***" multiplication is zero, the output is completely zeroed. Note the three sigmoids and the green ***tanh*** function all have weights and biases that need to be learned during training. This makes the LSTM more data intensive but also very powerful.

DDPG Neural Network Architecture

Now that all of the pieces have been discussed it is time to add them all together in the proposed DDPG network. Let's start by restating the state $\mathbf{S_t}$ as combination of the [3, n+1, k] price tensor $\mathbf{Y_t}$ and the [n+1] portfolio weighting $\mathbf{w_t}$. Recall that all n assets plus the S&P 500 index in $\mathbf{Y_t}$ has a low, high and close price for all k time embedding steps making a 3(n+1) x k tensor. In addition, $\mathbf{w_t}$ contains the portfolio weights of cash plus the n assets making a (n+1) vector. These are indicated by yellow in the figure below.

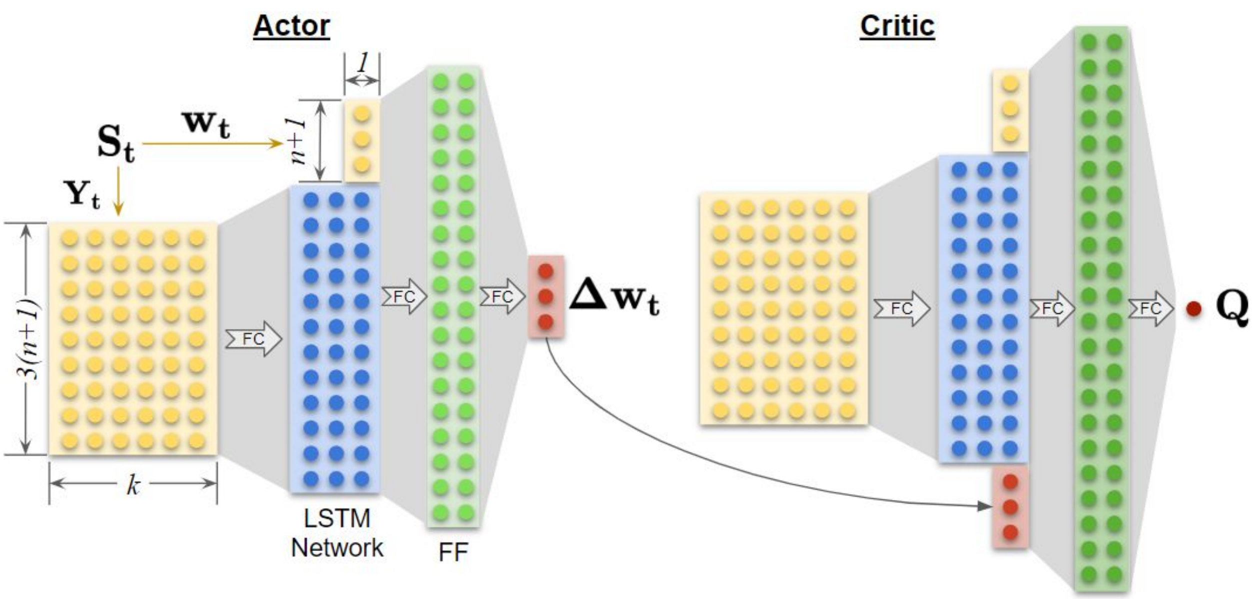


Figure 8. Proposed DDPG network diagram.

In the image above we see the price tensor $\mathbf{Y_t}$ is first passed to a LSTM network in both the actor and critic network. Note the depth and number of nodes in the LSTM network will be free parameters that we will need to experiment with in the next phase of this research. In the actor network, the output layer of the LSTM network is concatenated with the weight vector $\mathbf{w_t}$ and then passed to a standard feed-forward (FF) neural network with the final layer representing the predicted optimal action $\Delta\mathbf{w_t}$.

The critic network has a similar structure except the optimal action $\Delta\mathbf{w_t}$ generated by the actor is also concatenated with the LSTM output and the weight vector $\mathbf{w_t}$ before entering the FF network. The FF network now generates the **Q**-value of the state-action pair. Note the "FC" arrow in the diagram represents "fully-connected", which indicates that each node on the left is connected to every node on the right.

The actor and critic networks are trained independently and will have different weights and biases. For simplicity and considering the actor and critic LSTM networks see the same $\mathbf{Y_t}$ tensor, they will have the same structure. However the two FF networks see different input and generate an output layer with a different number of nodes, therefore they may have different structures. Testing will have to determine an optimal number of layers and number of nodes on each layer.

Both the actor and critic have local and target networks giving a total of four networks. The local and target networks have identical structures, function as discussed in the Double DQN section and will implement a soft update from the local to the target network.

Prioritized replay will be implemented in the architecture. It is believed that this will be very effective in training the network since particular market events may have a significant impact on the portfolio value; for both good and bad. Therefore it is important that these events are emphasized in the training.

Finally the network will incorporate an Ornstein-Uhlenbeck noise process directly on the network weighting as discussed in the OpenAI study.[28] This is a departure from the methodology covered in the Udacity Nanodegree but it appears to be a convenient way to boost performance.

[28] Matthias Plappert,et al. "Parameter Space Noise for Exploration." *OpenAI*, ICLR, 2016.

Training and Evaluation Process

We will take twelve years of daily price data and divide it into ten years of training and two years of evaluation. The length of these periods are relatively arbitrary and can be easily modified. However, periods need to be long enough to smooth out random motion and provide sufficient data for training. Ten years provide approximately 2530-k days of training. Other papers have been successful with amount of training data however only testing can determine if it is sufficient for the complexity of the network illustrated above.[29] We will have to experiment with the number of nodes and layers to find the appropriate complexity for the given data. We could extend the training period further into the past to obtain more data but we then have to consider if the market conditions are still applicable. If not, we may need to extend our state-definition to capture these macroeconomic effects.

Data Preparation

Prior to training we must screen/filter the price data. Ideally we would consider all of the assets that existed in the S&P 500 at any time over the 12 year training period. However this would make the proposed network to complex for the available training data. Consider figure 4 on page 14. Adding one more yellow input node adds new connections to each of the hidden nodes (four green nodes in figure 4). Each of these connections have weights which must be learned and as the number of inputs grows, the number of nodes also grows to capture more of the interactions. Now consider figure 8 on page 18; adding 1 more asset adds 4 more input nodes. For this reason the number of assets are often limited to a relatively small number such as 5 and 20 in studies by Liang and Hegde.[30,31]

The next filter performed by nearly all of the papers reviewed is to only consider assets with a full price history over the desired period. This is done to simplify the problem but does add a survivor bias to the experiment. Note we can randomly select the n assets from all of the assets that existed in the S&P 500 over the training period as long as price data is available. They do not have to be in the S&P 500 for the full period. Extending the process to include stocks that are listed or delisted during the period could be managed by setting their price to zero and forcing the weights to zero. Handling mergers and demergers would be more difficult. In the development of the

---

[29] Jason Brownlee. "How Much Training Data is Required for Machine Learning?" *Machine Learning Mastery*, 2017.

[30] Zhipeng Liang at al.. "Adversarial Deep Reinforcement Learning in Portfolio Management." *Sun Yat-sen University*, November 2018.

[31] Shashank Hegde et al.. "Risk aware portfolio construction using deep deterministic policy gradients," *IEEE SSCI*, 2018.

algorithm we will start with the simplest case; n randomly selected stocks that existed in the S&P 500 at the start of the period and with a full price history. This can then be extended to first new listing/delisting and then mergers/demergers as the system matures.

We must also adjust the high and low price data to compensate for splits, dividends and distributions.[32] Dividends are an interesting issue. A more realistic option would be to keep the dividend drops in the price data but also include the dividends in the reward function. However, the simpler price adjustment method will be used for this implementation.

Training

With clean data, we can begin training and pass through the training period, called an episode, many times. An episode continues from the beginning to the end of the training period unless the portfolio reaches zero, in which case a large negative reward is returned. Each of these episodes is analogous to a game if we were training the algorithm to play chess. The network retains the learned weights and biases from episode to episode allowing it to become "smarter" as it experiences more episodes. The memories defined by the $\langle \mathbf{s_t}, \mathbf{a_t}, \mathbf{r_{t+1}}, \mathbf{s_{t+1}}, \mathbf{v_t} \rangle$ tuples are also cumulative across all episodes, where $\mathbf{v_t}$ in the tuple is the priority of the memory. This allows us to accelerate learning by periodically resampling the memory to remember important events as the learning progresses.

Since each state includes the previous k-1 prices, each episode must begin after k-1 time steps. The network will also begin without any learning so the weights and bias will be initialized with the Xavier Initialization.[33] This initialization prevents the problem of vanishing or exploding gradients during the backpropagation in the training operation.

One of the major assumptions in this training process is that our actions do not impact the market. This allows us to reuse the asset prices across all episodes regardless of our actions. Including market impacts will be left for future work. However, if the network performs well in the remaining two-year evaluation period and someone was willing to risk real money, the training could continue with real actions. This would incorporate real market movements and transaction costs into the network.

---

[32] StockCharts.com. "Historical Price Data is Adjusted for Splits, Dividends and Distributions."
[33] Kian Katanforoosh and Daniel Kunin, "Initializing Neural Networks", *deeplearning.ai*, 2019.

Evaluation

With a trained network, we will then apply it to the final two years and compare its performance with respect to the overall return and Sharp ratio to the iShares Core S&P 500 ETF (IVV) including its 0.04% management fees. Note that during training the network never sees the two-year price data, however it can still learn as the two years progress. If the evaluation is repeated, the network is reset to the original trained state before it saw the evaluation data.

Implementation Environment

Now that the architecture and process are defined, we need to determine the implementation environment. The first part of that question is the software development environment. We have selected Python as the base software language due to its extreme popularity, ease of use and extensive machine learning packages.[34,35] From the many excellent Interactive Development Environments (IDE) available for Python we have selected PyCharm, which includes plug-ins for both GitHub and Amazon Web Service (AWS).[36]

The Python distribution we will use is Anaconda Python 3.7.[37] This is an industry standard that allows users to tailor their environment with Conda; an open source package management system.[38]

We are using the Git version control system and the GitHub code hosting platform. This not only provides a means of securing and controlling the code but also provides a collaboration platform to share this report and code and gather feedback from the community. GitHub also allows anyone to download (clone) the repository to any system including a local PC or AWS to test the code and provide updates.

[34] Nick Heath. "GitHub: The top 10 programming languages for machine learning." *TechRepublic*, Jan 2019.
[35] Mahesh Chand. "Best Programming Language for Machine Learning." C# Corner, Feb 2019.
[36] Jet Brains. "PyCharm: The Python IDE for Professional Developers." accessed Oct 26, 2019. https://www.jetbrains.com/pycharm/
[37] Anaconda. "Anaconda Distribution: The World's Most Popular Python/R Data Science Platform." accessed Oct 26, 2019. https://www.anaconda.com/distribution/
[38] Anaconda. "Conda: Package, dependency and environment management for any language." accessed Oct 26, 2019. https://conda.io/

PyTorch

With a general software development environment established we need to select the Machine Learning framework. The two main contenders currently are TensorFlow released by Google in 2015 and PyTorch by Facebook in 2017.[39,40] This is a rapidly evolving space and both packages are very capable and easily deployed with Python. Generally TensorFlow's TensorBoard visualization and production ready serving are advantages over PyTorch. PyTorch's advantages are a more pythonic style and dynamics graphing that make it easier to debug and develop.[41] For these reasons, we select PyTorch for development but the general strategy could be deployed in either framework.

Both PyTorch and TensorFlow also have built-in Graphics Processing Unit (GPU) support. GPUs make large deep learning networks practical to train.[42,43] When training a neural network there is first a forward pass that takes the input and propagates the values through the network to generate the output. The error is then calculated and back propagated through the network by calculating the gradient of the error with respect to each node weight, which is effectively a chain rule application. As the network becomes larger, these become extremely large matrix multiplication operations. These highly parallel matrix (tensor) operations are perfectly suited for GPUs. To automatically switch between CPU and GPU execution all we need to add is the following line in our PyTorch module.

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

For a full PyTorch implementation, please review the DDPG agent module in the tennis GitHub repository.[44] In this code the "`import torch`" line is importing the PyTorch package.

---

[39] TensorFlow. "An end-to-end open source machine learning platform." accessed Oct 26, 2019. https://www.tensorflow.org/

[40] PyTorch. "From Research To Production." accessed Oct 26, 2019. https://pytorch.org/

[41] Vihar Kurama. "Pytorch Vs. Tensorflow: Which Framework Is Best For Your Deep Learning Project?" *Builtin*, Sept 2019.

[42] Bernard Fraenkel. "For Machine Learning, It's All About GPUs." *Forbes*, Dec 2017.

[43] Shachi Shah. "Do we really need GPU for Deep Learning? - CPU vs GPU." *Medium.com*, Mar 2018.

[44] Daniel Fudge. "ddpg.py: DRL applied to Tennis Environment." accessed Oct 26, 2019. https://github.com/daniel-fudge/reinforcement-learning-tennis/blob/master/tennis/ddpg_agent.py

Amazon Web Service (AWS)

The second major component of the environment is the hardware. As discussed above, deploying this algorithm on GPUs is a major speed boost that makes testing and debugging much faster. Aside from buying GPU hardware, there are many cloud computing environments available that provide GPU support. The environment we have selected is AWS. It's Deep Learning Amazon Machine Images (DLAMIs) with Conda allows us to rapidly launch the latest GPU enabled hardware with PyTorch and all necessary software pre-installed.[45] This greatly simplifies the setup and management of the environment and eliminates the need for new users go through a lengthy and complicated setup procedure.

During the Udacity DRL Nanodegree the p3.2xlarge Ubuntu 16.04 DLAMI with one GPU was used to train and evaluate the tennis DDPG network.[46,47] Due to the limited training data size and small network, the switch from CPU to GPU wasn't significant. However we expect this to be more important for the much larger network we will be training.

[45] AWS. "AWS Deep Learning AMIs." accessed Oct 26, 2019. https://aws.amazon.com/machine-learning/amis/

[46] AWS Marketplace. "Deep Learning AMI (Ubuntu 16.04)." accessed Oct 26, 2019. https://aws.amazon.com/marketplace/pp/B077GCH38C

[47] Daniel Fudge. "DRL applied to Tennis Environment." accessed Oct 26, 2019. https://github.com/daniel-fudge/reinforcement-learning-tennis

Future Research Prior to Implementation

AWS Development Environment

Testing in the AWS environment included a local laptop development then transfer to a DLAMI that cost $3.06/hr for on-demand usage. Another option is the AWS Cloud9 development environment.[48] Developing on AWS allows live debugging with GPUs and presents a fixed development environment for multiple developers. However, we do not want to be charged $3.06/hr for development that does not include training. One option is to split the development and training environments. For instance if you spend 90 hours per month developing, the monthly charge for a t2.micro instance with 10 GB of storage would be $2.05, which is the same cost of 40 minutes on the DLAMI.[49] However, you would not want to train in this instance, so you need to dispatch training to the DLAMI. To reduce the cost up to 90%, spot pricing may be used for the DLAMI.[50] This basic idea of spot pricing is you submit a request for a resource at a given price. Your job will not start until the “Spot Price”, which is set by AWS based on supply and demand for that specific resource, drops below your requested price. If the price rises above your price while your are running, the instance will be terminated. This is acceptable in our workflow as long as we periodically save our network weights. Training can restart from the saved weights once the price drops again.

A different approach is using the AWS SageMaker.[51] However it isn’t clear if this would provide much benefit for custom PyTorch implementations designed for research instead of production workflows. The Pluralsight “Using PyTorch in the Cloud: PyTorch Playbook” course includes a section devoted to deploying PyTorch deep learning models on AWS SageMaker.[52] The “AWS Certified Machine Learning - Specialty 2019” by A Cloud Guru also covers the broader implementation of SageMaker.[53]

Understanding the optimal development environment within AWS is future research that must be performed before implementation begins. The documentation of this development environment would be captured in the next term report and will lower the barrier to entry for new researchers.

---

[48] AWS, “AWS Cloud9.” accessed Oct 26, 2019. https://aws.amazon.com/cloud9/
[49] AWS, “Cloud9 Pricing.” accessed Oct 26, 2019. https://aws.amazon.com/cloud9/pricing/
[50] AWS, “Amazon EC2 Spot Instances.” accessed Oct 26, 2019. https://aws.amazon.com/ec2/spot/
[51] AWS, “Reinforcement Learning with Amazon SageMaker RL.” accessed Oct 26, 2019. https://docs.aws.amazon.com/sagemaker/latest/dg/reinforcement-learning.html
[52] Pluralsight. “Using PyTorch in the Cloud: PyTorch Playbook.” accessed Oct 26, 2019. https://www.pluralsight.com/courses/using-pytorch-in-the-cloud-playbook
[53] A Cloud Guru. “AWS Certified Machine Learning - Specialty 2019.” accessed Oct 26, 2019. https://acloud.guru/learn/aws-certified-machine-learning-specialty

LSTM Network Architectures

The DDPG networks developed for this research did not include a LSTM network. The theory and basic implementation of a LSTM network are described in the preceding sections, however a true understanding of how to size and connect a LSTM network can only be obtained through implementing such a network. The Udacity "Intro to Deep Learning with PyTorch" course includes building a LSTM network in PyTorch and could provide the necessary experience.[54] This course could also be extended to the Udacity "Deep Learning" Nanodegree.[55] Coursera also has a "Advanced Machine Learning with TensorFlow on Google Cloud Platform Specialization" that appears very useful but it uses TensorFlow instead of PyTorch and Google Cloud instead of AWS as is used in Udacity.

Future Work Beyond Implementation

Increasing State Definition

The state definition we have included here is purely dependent on the prices of the assets in the portfolio and a market index. Clearly this could be expanded to incorporate a multitude of signals. This could include macroeconomic signals such as Gross Domestic Product (GPD), inflation rates, interest rates and unemployment rates. It could also include firm specific financial indicators such as liquidity, efficiency, profitability and leverage ratios. More creative signals could be added such as sentiment analysis of twitter feeds or news reports or processing of satellite imagery. Selecting which signals to include and how to pre-process this data is a critical area of research.

The Udacity Artificial Intelligence (AI) for Trading Nanodegree may be relevant to this research.[56] It focuses on using AI to generate trading signals from a wide variety of sources. These signals could be added to the state definition for the DRL algorithm.

Increasing Transaction Cost Accuracy

Our representation of the transaction cost is simply a constant times the change in portfolio weighting. Clearly this could be replaced by a more accurate cost model.

---

[54] Udacity. "Free Course: Intro to Deep Learning with PyTorch." accessed Oct 26, 2019. https://www.udacity.com/course/deep-learning-pytorch--ud188.

[55] Udacity. "Nanodegree Program: Deep Learning." accessed on Oct 26, 2019. https://www.udacity.com/course/deep-learning-nanodegree--nd101

[56] Udacity. "Nanodegree Program: Artificial Intelligence for Trading." accessed Oct 26, 2019. https://www.udacity.com/course/ai-for-trading--nd880

Including Market Impact of Trades

As discussed in the state-space definition on page 8, we selected assets from the S&P 500 so that we could assume that our trades would not impact the asset prices. This is a reasonable assumption as long as our trades are a small fraction of the market. However, adding these market impacts would be a reasonable upgrade to the algorithm.

Allowing for Dynamic Selection of Assets

Our previous discussion of data preparation highlighted three short-comings of most papers related to DRL for portfolio optimization. The first is that they consider only a limited number of assets where as a human managing the portfolio would be monitoring many more assets. I don't believe simply adding more assets to the existing network architectures is viable but there many be ways to segregate the assets into a more efficient network for large numbers of assets.

Second is once the assets are selected the portfolio membership is fixed forever. Again this doesn't seem realistic. Similar to a human managing the portfolio, there could be a higher level algorithm that learns to replace assets based on possibly a different state space. This algorithm could also incorporate newly listed and delisted stocks.

The third short-coming was the inability to deal with mergers, demergers and other price discontinuities. This seems like a tractable problem with the addition of the proper reward function and a function to reprocess the state-space and LSTM matrix.

Implementing more Advanced DRL Architectures

The field of DRL is constantly evolving and future work should compare our DDPG algorithm to many of the competing algorithms such as OpenAI's Proximal Policy Optimization (PPO).[57] This effort should be expanded as more algorithms are published and this framework could be used to experiment with new algorithms and concepts.

In our architecture we employ a LSTM to capture the time history of the asset prices, however this is also a very active field that needs to be explored. Many of the methods try to incorporate the concepts of Convolutional Neural Networks (CNN), which are most often deployed in image recognition.[58,59] In this context they are called Temporal

---

[57] John Schulman et al.. "Proximal Policy Optimization Algorithms." OpenAI, August 28, 2017.

[58] Taewook Kim et al.. "Forecasting stock prices with a feature fusion LSTM-CNN model using different representations of the same data." *PLoS One*, Feb 2019.

[59] Shaojie Bai et al.. "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling." *ArXiv*, April, 2018.

Convolutional Networks (TCN). Another extension to CNNs adds dilation, where the output of a CNN is fed into a subsequent CNN that samples at a lower frequency as shown below with a stride and dilation factor of one.[60] This should be much more memory efficient with minimal loss of information and can also be applied to LSTM.[61]

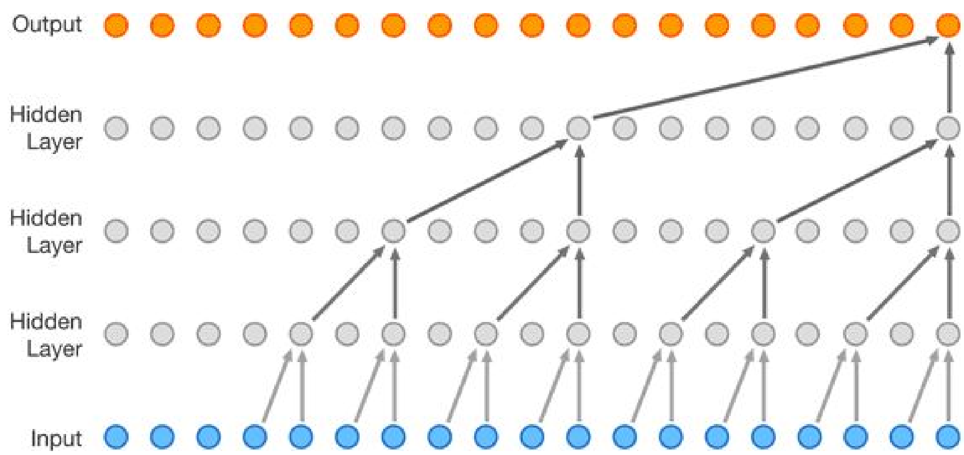


Figure 9. Dilated convolutional layers.
Image Source: https://i.stack.imgur.com/OCyiq.gif

[60] Fisher Yu and Vladlen Koltun. "Multi-scale Context Aggregation By Dilated Convolutions." ICLR, 2016.
[61] Shiyu Chang et al.. "Dilated Recurrent Neural Networks." *NIPS 2017*, April 19, 2018.

Conclusion

This report summarized DRL concepts related to our proposed problem statement and proposed a DDPG architecture for portfolio optimization. It also detailed a PyTorch development environment and its deployment on AWS to leverage the power of GPUs on the AWS DLAMI. For this implementation it also identified research that must be executed prior to its implementation and future work to extend the implementation.

This report in concert with the Udacity Nanodegree in DRL concludes the 2nd term Independent Study into Machine Learning applications to portfolio optimization. The 1st term was a general investigation into the "Application of Machine Learning to Portfolio Optimization". Here we reviewed the different aspects of machine learning and their possible applications to Portfolio Optimization. During this investigation we highlighted Reinforcement Learning (RL) as an especially promising area to research and proposed the development of a Reinforcement Learning framework to better understand its possible applications.

With the problem statement and architecture defined in this report, it will be the task of the 3rd term to fully implement the architecture. Note that the intent of this implementation is not to advance the state-of-the-art in DRL or its application to portfolio optimization. Instead it is to generate a functioning DRL platform for portfolio optimization. From this conventional implementation, we can experiment with more advanced techniques as highlighted in this report. Term 3 will also deliver a report that summarizes not only the implementation but also the development environment to lower the barrier to entry for researchers new to DRL and the deployment of cloud-based applications.

The combination of all three terms and the associated code can be found in GitHub repository linked below.

https://github.com/daniel-fudge/DRL-Portfolio-Optimization

Bibliography

Bai, Shaojie, J. Zico Kolter and Vladlen Koltun. “An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling.” *ArXiv*, April 19, 2018. https://arxiv.org/abs/1803.01271

Brownlee, Jason. “How Much Training Data is Required for Machine Learning?” Machine Learning Mastery, July 24, 2017. https://machinelearningmastery.com/much-training-data-required-machine-learning/

Chand, Mahesh. “Best Programming Language for Machine Learning.” *C# Corner*, February 2, 2019. https://www.c-sharpcorner.com/article/best-programming-language-for-machine-learning/

Chang, Shiyu, Yang Zhang, Wei Han, Mo Yu, Xiaoxiao Guo, Wei Tan, Xiaodong Cui, Michael Witbrock, Mark Hasegawa-Johnson, Thomas S. Huang. “Dilated Recurrent Neural Networks.” *31st Conference on Neural Information Processing Systems (NIPS 2017)*, Long Beach, CA, USA, April 19, 2018. https://arxiv.org/abs/1803.01271

Doob, J.T.. “The Brownian Movement and Stochastic Equations.” *Annals of Mathematics*, Second Series, Vol. 43, No. 2, pp. 351-369, April 1942. https://www.jstor.org/stable/1968873

Filos, Angelos. “Reinforcement Learning for Portfolio Management.” MEng Diss., Imperial College London, June 20, 2018. https://arxiv.org/abs/1909.09571

Fraenkel, Bernard. “For Machine Learning, It's All About GPUs.” *Forbes*, December 1, 2017. https://www.forbes.com/sites/forbestechcouncil/2017/12/01/for-machine-learning-its-all-about-gpus/#45e138417699

Fudge, Daniel. “DRL-Portfolio-Optimization.” *GitHub Project*. https://github.com/daniel-fudge/DRL-Portfolio-Optimization

Fudge, Daniel. “Application of Machine Learning to Portfolio Optimization.” Independent Study for course FINE6900 (with Professor Larkin), *Schulich School of Business*, April 2019. https://github.com/daniel-fudge/DRL-Portfolio-Optimization/blob/master/docs/report1.pdf

Graves, Alex, Santiago Fernandez and Jurgen Schmidhuber. “Multi-Dimensional Recurrent Neural Networks.” *IDSIA*, 2013. https://arxiv.org/abs/0705.2011

Greff, Klaus, Rupesh K. Srivastava, Jan Koutnık, Bas R. Steunebrink and Jurgen Schmidhuber. “LSTM: A Search Space Odyssey.” *IEEE Transactions on Neural Networks and Learning Systems,* Vol. 28, No. 10, pp. 2222-2232, October 2017. https://arxiv.org/abs/1503.04069

Gu, Shixiang, Timothy Lillicrap, Ilya Sutskever and Sergey Levine. "Continuous Deep Q-Learning with Model-based Acceleration." *Google Brain and Google DeepMind*, March 2016. https://arxiv.org/abs/1603.00748

Halperin, Igor. "QLBS: Q-Learner in the Black-Scholes(-Merton) Worlds." *SSRN Electronic Journal*, (December 2017) DOI: 10.2139/ssrn.3087076. https://arxiv.org/abs/1712.04609

Hasselt, Hado van, Arthur Guez and David Silver. "Deep Reinforcement Learning with Double Q-Learning." *Google DeepMind*, 13th Conference on Artificial Intelligence (AAAI-16), 2016. https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/viewPaper/12389

Heath, Nick. "GitHub: The top 10 programming languages for machine learning." *TechRepublic*, January 25, 2019. https://www.techrepublic.com/article/github-the-top-10-programming-languages-for-machine-learning/

Hegde, Shashank, Vishal Kumar and Atul Singh, "Risk aware portfolio construction using deep deterministic policy gradients," *IEEE Symposium Series on Computational Intelligence (SSCI)*, 2018, pp. 1861-1867. doi: 10.1109/SSCI.2018.8628791. https://ieeexplore.ieee.org/document/8628791

Hessel, Matteo, Will Dabney, Joseph Modayil, Dan Horgan, Hado van Hasselt, Bilal Piot, Tom Schaul, Mohammad Azar, Georg Ostrovski, David Silver. "Rainbow: Combining Improvements in Deep Reinforcement Learning." *Google DeepMind*, Association for the Advancement of Artificial Intelligence (AAAI), 2018. https://arxiv.org/abs/1710.02298

Hochreiter, Sepp and Jurgen Schmidhuber "Long short-term memory." *Neural computation*, 9(8):1735–1780, 1997. https://www.mitpressjournals.org/doi/abs/10.1162/neco.1997.9.8.1735

Hui, Jonathan. "RL-Value Learning." *Medium*, October 11, 2018. https://medium.com/@jonathan_hui/rl-value-learning-24f52b49c36d

Jiang, Zhengyao, Dixing Xu and Jinjun Liang. "A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem." *Journal of Machine Learning Research (JMLR)*, July 2017. https://arxiv.org/abs/1706.10059

Katanforoosh, Kian and Daniel Kunin, "Initializing Neural Networks", *deeplearning.ai*, 2019. https://www.deeplearning.ai/ai-notes/initialization/

Kim, Taewook, Ha Young Kim and Raul Harnandez Montoya. "Forecasting stock prices with a feature fusion LSTM-CNN model using different representations of the same data." *PLoS One*, February 15, 2019. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6377125/

Konda, Vijay R. and John N. Tsitsiklis. "Actor-Critic Algorithms." *Massachusetts Institute of Technology*, Cambridge, 2000. http://papers.nips.cc/paper/1786-actor-critic-algorithms.pdf

Kurama, Vihar. "Pytorch Vs. Tensorflow: Which Framework Is Best For Your Deep Learning Project?" *Builtin*, September 22, 2019. https://builtin.com/data-science/pytorch-vs-tensorflow

Liang, Zhipeng, Hao Chen, Junhao Zhu, Kangkang Jiang, Yanran Li. "Adversarial Deep Reinforcement Learning in Portfolio Management." *Sun Yat-sen University*, November 2018. https://arxiv.org/abs/1808.09940

Lillicrap, Timothy, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver and Daan Wierstra. "Continuous Control With Deep Reinforcement Learning." *Proceedings from the International Conference on Learning representations (ICLR)*, 2016. https://arxiv.org/pdf/1509.02971.pdf

Olah, Christopher. "Understanding LSTM Networks." *colah's blog*, August 2015. http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Peters, Jan, Sethu Vijayakumar and Stefan Schaal. "Natural Actor Critic." *Neurocomputing*, Volume 71, Issues 7-9, March 200 https://doi.org/10.1016/j.neucom.2007.11.026

Plappert, Matthias, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Tamim Asfour, Pieter Abbeel and Marcin Andrychowicz. "Parameter Space Noise for Exploration." *OpenAI*, Proceedings from the International Conference on Learning representations (ICLR), 2016. https://arxiv.org/abs/1706.01905

Plappert, Matthias, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Tamim Asfour, Pieter Abbeel and Marcin Andrychowicz. "Parameter Space Noise for Exploration." *OpenAI (Blog)*, 2017. https://openai.com/blog/better-exploration-with-parameter-noise/

Ross, Stephen A., Randolph W. Westerfield, Bradford D. Jordan and Gordon S. Roberts. "The Efficient Markets Hypothesis" in *Fundamentals of Corporate Finance*, 9th edition, McGraw-Hill Education, 2019.

Santos, Leonardo Araujo dos. "Artificial Intelligence." GitBook, accessed October 6, 2019. https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/

Schaul, Tom, John Quan, Ioannis Antonoglou and David Silver. "Prioritized Experience Replay." *Google DeepMind*, International Conference on Learning Representations (ICLR) 2016. https://arxiv.org/abs/1511.05952

Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford and Oleg Klimov. "Proximal Policy Optimization Algorithms." *OpenAI*, August 28, 2017. https://arxiv.org/abs/1707.06347

Schulman, John, Sergey Levine, Phillipp Moritz, Michael Jordon and Pieter Abbeel. "Trust Region Policy Optimization." *University of California*, Berkeley, 31st International Conference on Machine Learning, 2015. https://arxiv.org/abs/1502.05477

Shah, Shachi. "Do we really need GPU for Deep Learning? - CPU vs GPU." *Medium.com*, March 26, 2018. https://medium.com/@shachishah.ce/do-we-really-need-gpu-for-deep-learning-47042c02efe2

Siblis Research. "S&P 500 Historical Total Market Cap & Float Adjusted Cap." September 30, 2019. http://siblisresearch.com/data/total-market-cap-sp-500/

Silver, David, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra and Martin Riedmiller. "Deterministic Policy Gradient Algorithms." *Proceedings of the 31 st International Conference on Machine Learning*, Beijing, China, 2014. http://www.jmlr.org/proceedings/papers/v32/silver14.pdf

StockCharts.com. "Historical Price Data is Adjusted for Splits, Dividends and Distributions." Accessed Oct 25, 2019. https://support.stockcharts.com/doku.php?id=policies:adjusted_data

Sutton, Richard S. and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd edition, Cambridge, Massachusetts: The MIT Press, A Bradford Book, (November 5 2018). http://incompleteideas.net/book/RLbook2018.pdf

Udacity. "Deep Reinforcement Learning." Nanodegree program. https://www.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893

Uhlenbeck, George E. and Leonard S. Ornstein. "On the Theory of the Brownian Motion." *Physical review*, 36(5): 823, 1930. https://doi-org.ezproxy.library.yorku.ca/10.1103/PhysRev.36.823

Wawrzynski, Pawel. "Real-time reinforcement learning by sequential Actor-Critics and experience replay." *Neural networks*: the official journal of the International Neural Network

Society. 22. 1484-97 (June 2019). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.904.8961&rep=rep1&type=pdf

Williams, Ronald J. "Simple statistical gradient-following algorithms for connectionist reinforcement learning." Reinforcement Learning. Springer, Boston, MA, 1992.5-32.

Yang, Qing, Tingting Ye and Liangliang Zhang. "A General Framework of Optimal Investment." *SSRN* (February 2, 2019). https://ssrn.com/abstract=3136708

Yu, Fisher and Vladlen Koltun. "Multi-scale Context Aggregation By Dilated Convolutions." International Conference on Learning Representations (ICLR), April 30, 2016. https://arxiv.org/abs/1511.07122

Yu, Pengqian, Joon Sern Lee, Ilya Kulyatin, Zekun Shi and Sakyasingha Dasgupta. "Model-based Deep Reinforcement Learning for Financial Portfolio Optimization." *arXiv:1901.08740* (January 2019). https://arxiv.org/abs/1901.08740

Appendix A - Related Training

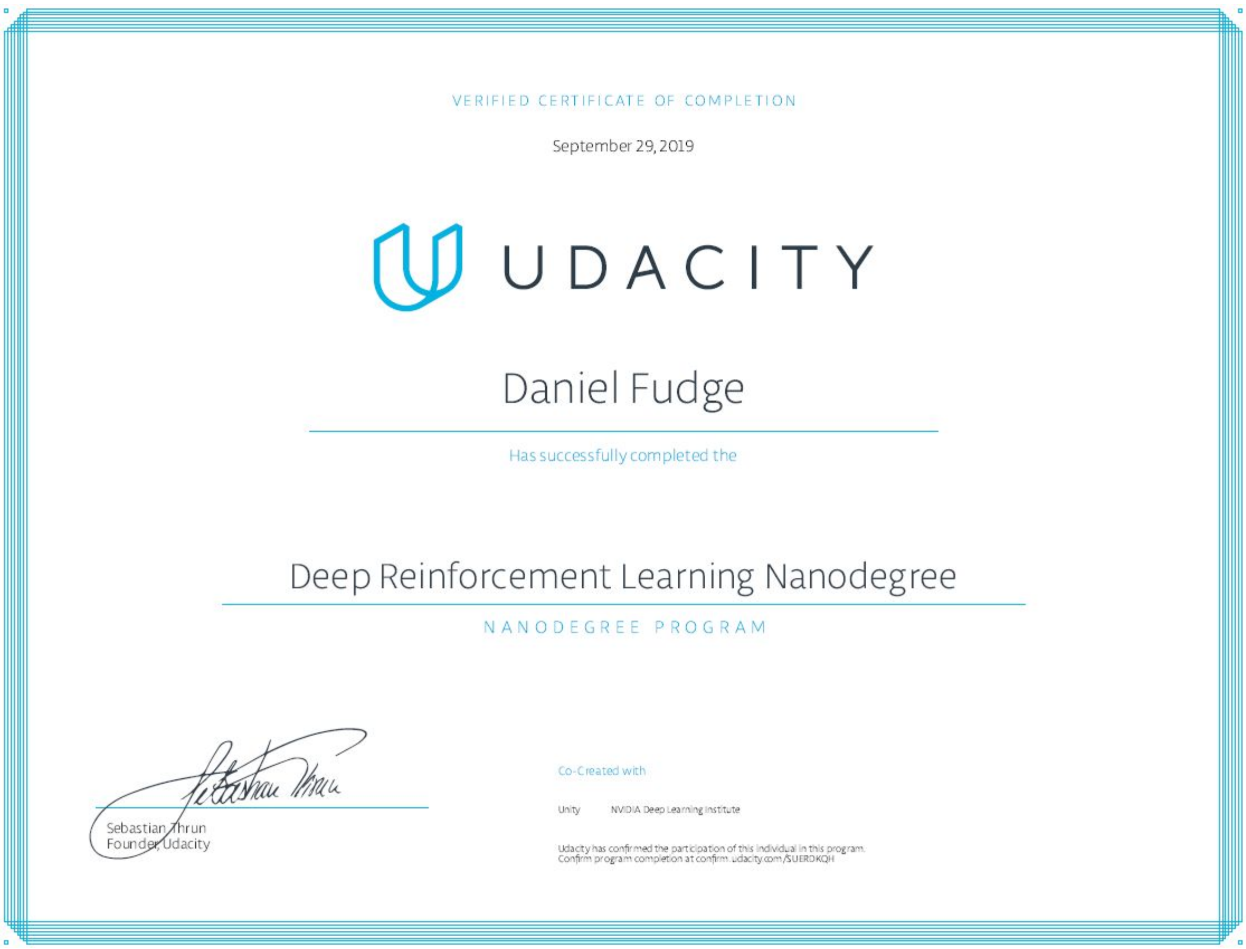
A.1 Udacity Nanodegree: Deep Reinforcement Learning (DRL)

https://www.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893

This Nanodegree involved developing DRL networks to solve 3 different Unity ML-Agents environments (https://unity3d.com/machine-learning/).

The solutions to these environment can be found in separate GitHub repositories:

1. Banana Collector: https://github.com/daniel-fudge/banana_hunter
2. Reacher: https://github.com/daniel-fudge/reinforcement-learning-reacher
3. Tennis: https://github.com/daniel-fudge/reinforcement-learning-tennis


VERIFIED CERTIFICATE OF COMPLETION

September 29, 2019

UDACITY

Daniel Fudge

Has successfully completed the

Deep Reinforcement Learning Nanodegree

NANODEGREE PROGRAM

Sebastian Thrun
Founder, Udacity

Co-Created with

Unity NVIDIA Deep Learning Institute

Udacity has confirmed the participation of this individual in this program.
Confirm program completion at confirm.udacity.com/SUERDKQH