Deep Learning Project Report

Deep Deterministic Policy gradient for Portfolio Optimisation

---

CentraleSupélec

Diallo Alassane, Mahamadou Dia

Ecole CentraleParis
France
13/12/20

Abstract

The field of reinforcement learning has experienced huge progress during the past years with the combination of deep learning techniques coupled with Reinforcement learning resulting in Deep reinforcement learning algorithms. These algorithms are very practical when dealing with real world problems where the degree of freedom of the environment is huge.
A famous reinforcement learning method, deep Q network recognised for its ability to model Q value function, has shown some limitations. In fact the way deep Q network works is that it discretizes the action space which makes the complexity exponential in case of continuous action space. This is where Deep reinforcement methods come to play to correct the complexity issue.
My work in this paper would be twofold. First to summarize quickly the work of the original paper "A deep reinforcement learning framework for portfolio management problem" . And secondly to present my modification of the algorithm and compare its performance to the original one .

keywords = Deep learning, reinforcement learning, stochastic control, portfolio management.

Contents

Reinforcement learning

We will start by introducing the concept of reinforcement learning and more specifically the deep reinforcement learning algorithm. This will be useful for the next section.

We consider a standard reinforcement learning setup consisting of an Agent interacting with an Environment E (which may be stochastic) in discrete time steps . At each time step, the agent receives an observation x_t, takes an action a_t an receives a scalar reward r_t. We suppose the action to take its values in $\mathcal{R}^N$. To describe a state s_t of an environment, we generally consider a partial observation of the environment so as $s_t = (x_1, a_1, ..., a_{t-1}, x_t)$. Here we assumed the environment is fully-observed so $s_t = x_t$. An agent's behavior is defined by a policy π, which maps states to a probability distribution over actions , $\pi : \mathcal{S} \leftarrow \mathcal{P}$. Since the environment can be stochastic , we model it as a Markov decision process with a state space $\mathcal{S}$, action space $\mathcal{A} = \mathcal{R}^N$ an initial state distribution $p(s_1)$, transition dynamics $p(s_{t+1}/s_t, a_t)$ and reward function $r(s_t, a_t)$. The return from a state is defined as the sum of discounted future reward $R_t = \sum_{i=t}^{T} \gamma^{(i-t)} r(s_i, a_i)$ with a discount factor $\gamma \in [0, 1]$. Since the return is dependent upon the action chosen , and therefore on the policy π, it may be stochastic. The goal of reinforcement learning is to learn a policy which maximises the expected return from the start distribution $\mathrm{J} = \mathrm{E}_{r_i, s_i \sim E, a_i \sim \pi}[R_t | s_t, a_t]$. We denote the discounted state visitation distribution for a policy as ρ^π.

The action-value function is used in many reinforcement learning algorithms. It describes the expected return after taking an action a_t, in state s_t and thereafter following policy π:

$$Q^\pi(s_t, a_t) = \mathrm{E}_{r_{i \geq t},\ s_{i \geq t} \sim \mathrm{E}}[R_t | s_t, a_t] \qquad (1)$$

Many approaches in reinforcement learning make use of the recursive relationship known as the **bellman equation**:

$$Q^\pi(s_t, a_t) = E_{r_t, s_{t+1} \sim E}[r(s_t, a_t) + \gamma E_{a_{t+1} \sim \pi}[Q^\pi(s_{t+1}, \mu(s_{t+1}))]] \qquad (2)$$

If the target policy π is deterministic,that is for a given state the action to take can be represented by the function $\mu : \mathcal{S} \leftarrow A$. we can introduce it to the previous equation and avoid the inner expectation :

$$Q^\mu(s_t, a_t) = E_{r_t, s_{t+1}}[r(s_t, a_t) + \gamma Q^\mu(s_{t+1}, \mu(s_{t+1}))] \qquad (3)$$

The expectation depends only on the environment. This means that it is possible to learn Q^μ independently from the action policy(which is called off-policy method).

To solve this optimisation problem we introduce a function approximators parameterized by θ^Q, which we optimize by minimizing the loss:

$$L(\theta^Q) = E_{s_t \sim \rho^\beta, a_t \sim \beta, r_t \sim E}[(Q(s_t, a_t|\theta^Q) - y_t)^2]$$

where

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta^Q) \qquad (5)$$

In the next section we will present a reinforcement learning algorithm used to solve such optimisation problem namely Deep deterministic policy gradient.

1.1 The Deep deterministic policy gradient

It is not possible to straightforwardly apply Q-learning to continuous action spaces because in continuous spaces finding the greedy policy requires an optimization of a_t at every timestep. This optimization is to slow to be practical to non trivial action spaces. Instead we used an actor-critic approach based on the Deep Policy Gradient algorithm. The DPG algorithm maintains a parameterized actor function $\mu(s|\theta^\mu)$ which specifies the current policy by deterministically mapping states to a specific action. The critic Q(s,a) is learned using the Bellman equation as in Q-learning. The actor is updated by following the chain rule to the expected return from the start distribution J with respect to the actor parameters:

$$\begin{aligned}\nabla_{\theta^u} J &\approx E_{s_t \sim \rho^\beta}[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}] \\ &= E_{s_t \sim \rho^\beta}[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)\nabla_{\theta^\mu}\mu(s|\theta^\mu)|s=s_t}]\end{aligned}$$

As with Q learning, introducing non-linear function approximators means that convergence is no longer guaranteed .However, such approximators appear essential in order to learn and generalize on large state spaces. The idea here is to provide modifications to DPG which allow it to use neural network function approximators to learn in large state and action spaces online.

One challenge when using neural networks for reinforcement learning is that most optimization algorithms assume that the samples are independently and identically distributed. Obviously when the samples are generated from exploring sequentially in an environment this assumption no longer holds. Additionally, to make use of hardware optimizations, it is essential to learn in mini batches rather than online.

As in DQN, we used a replay buffer to address the issues. The replay buffer is a finite sized cache $\mathcal{R}$. Transition were sampled from the environment according to the exploration policy and the tuple (s_t, a_t, r_t, s_{t+1}) was stored in the replay buffer. When the replay buffer was full the oldest samples were discarded.At each time step, the actor critic are updated by sampling a mini batch uniformly from the buffer. Because DDPG is an off-policy algorithm , the replay buffer can be large , allowing the algorithm to benefit from learning across a set of uncorrelated transition.

Directly implementing Q learning (equation 4) with neural networks proved to be unstable in many environments. Since the network $Q(s, a|^Q)$ being updated is also used in calculating the target value (equation 5), the Q update is prone to divergence. We create a copy of the actor and critic networks $\mu'(s|\theta^{\mu'})$, $Q(s, a|\theta^Q)$ respectively, that are used for calculating the target values. The weights of these target networks are then updated by having them slowly track the learned networks:

$_{,}\tau\theta + (1-\tau)\theta'$ with $\tau \ll 1$. This means that the target values are constrained to change slowly, greatly improving the stability of learning.

When learning from low dimensional feature vector observations, the different components of the observation may have different physical units (for example, positions versus velocities) and the ranges may vary across environments. This can make it difficult for the network to learn effectively and may make it difficult to find hyper-parameters which generalise across environments with different scales of state values. One approach to this problem is to manually scale the features so they are in similar ranges across environments and units. We address this issue by adapting a recent technique from deep learning called batch normalization . This technique normalizes each dimension across the samples in a minibatch to have unit mean and variance. In addition, it maintains a running average of the mean and variance to use for normalization during testing (in our case, during exploration or evaluation). In deep networks, it is used to minimize covariance shift during training, by ensuring that each layer receives whitened input. In the low-dimensional case, we used batch normalization on the state input and all layers of the network and all layers of the Q network prior to the action input (details of the networks are given in the supplementary material). With batch normalization, we were able to learn effectively across many different tasks with differing types of units, without needing to manually ensure the units were within a set range.

A major challenge of learning in continuous action spaces is exploration. An advantage of off- policies algorithms such as DDPG is that we can treat the problem of exploration independently from the learning algorithm. We constructed an exploration policy by adding noise sampled from a noise process $\mathcal{N}$ to our actor policy :

$$\mu'(s_t|\theta_t^\mu) + \mathcal{N} \qquad (7)$$

$\mathcal{N}$ can be chosen to suit the environment. As detailed in the supplementary materials we used an Ornstein-Uhlenbeck process to generate temporally correlated exploration for exploration efficiency in physical control problems with inertia.

In the next section we will dive into the Deep Reinforcement Learning Paper. We will explain the main points presented therein.

Deep Reinforcement Learning

The original paper presents a deep deterministic policy gradient methods to solve portfolio allocation problem. In simple term, we want to allocate a portfolio consisting of 11 cryptocurrencies taken from the Poloniex Exchange.The performance of the Algorithm is after compared to that of existing asset allocation strategies which I will present later.
Before diving into further detail, Let's analyse the cryptocurrency market.

2.1 Cryptocurrency market

Cryptographic currencies also called cryptocurrencies is a decentralized electronic alternatives to government issues money . Though Bitcoin remains the most renowned cryptocurrency, there are more than 100 tractable other cryptocurrencies competing each other and with Bitcoin. The motive of this competition is that Bitcoin has a lot of design flaws and people are creating more new coins to overcome those defects hoping that their invention would replace Bitcoin .There are , however, more and more cryptocurrencies being created that do not target to beat bitcoin . but with the purposes of using the technology behind it to develop decentralized applications.

Two natures of cryptocurrencies differentiate them from the other financial assets and ultimately make them the most test-grounded for portfolio management experiments. These natures are decentralization and openness, and the former implies the latter. Because Bitcoin is decentralised there is no low entrance requirement which means that anyone can participate into it . One direct consequence is abundance of small volume currencies. Affecting the prices of these penny markets will require smaller amount of investment, compared to traditional markets. Openness characteristic also means that markets are more accessible. Most cryptocurrencies exchanges have application programming interface for obtaining market data and carried out trading actions. Also most exchanges are open 24/7 without restricting the frequency of tradings. This non stop markets are ideal for machines to learn in the real world in shorter time-frames.

Now let's talk about the Formulation of the problem into a reinforcement learning Language.To do that we will define the Environment(set of different states), the Agent and the action taken by the Agent in a given state.

2.2 Trading Environment

The Environment is where the Agent evolves and interacts with .In our trading case it corresponds to the stocks data and the history of all the possible positions taken by the Agent.

2.3 Trading Agent

Our trading Agent is the Deep Reinforcement learning Algorithm evolving in an Environment which will be characterised later. The goal of the Agent is to maximise its overall reward. The reward here corresponds to the overall profit of the portfolio. In other words our Agent will dynamically allocate its weighted vectors so to maximise the its portfolio return. By setting R as the cumulative logarithmic return, we can write it as :

$$R(s_1, a_1, ..., s_{t_f}, a_{t_f}, s_{t_f+1}) := \frac{1}{t_f} \ln \frac{p_f}{p_0} = \frac{1}{t_f} \sum_{t=1}^{t_f+1} \ln(\mu_t y_t . w_{t-1})$$

$$= \frac{1}{t_f} \sum_{t=1}^{t_f+1} r_t$$

where p_f is the final price of the asset and p_0 the initial price.

Now that the different variables have being clearly define , the next steps is to use data to train the trading model.

2.4 Presentation of the paper results

TO train the trading algorithm to perform return optimisation task, data had been collected in an cryptocurrencies Exchange market called Poloniex Exchange. The portfolio is composed of 11 cryptocurrencies and one benchmark(bitcoin). The data had been processed and put in a specified Deep neural Neural network representing the architecture of the policy gradient (defined in previous chapters). The figure below represents one of the three architectures of the policy gradient network proposed in the paper namely CNN, RNN, LSTM . Our chosen one is built on the basis of the Convolutional Neural Network(CNN).

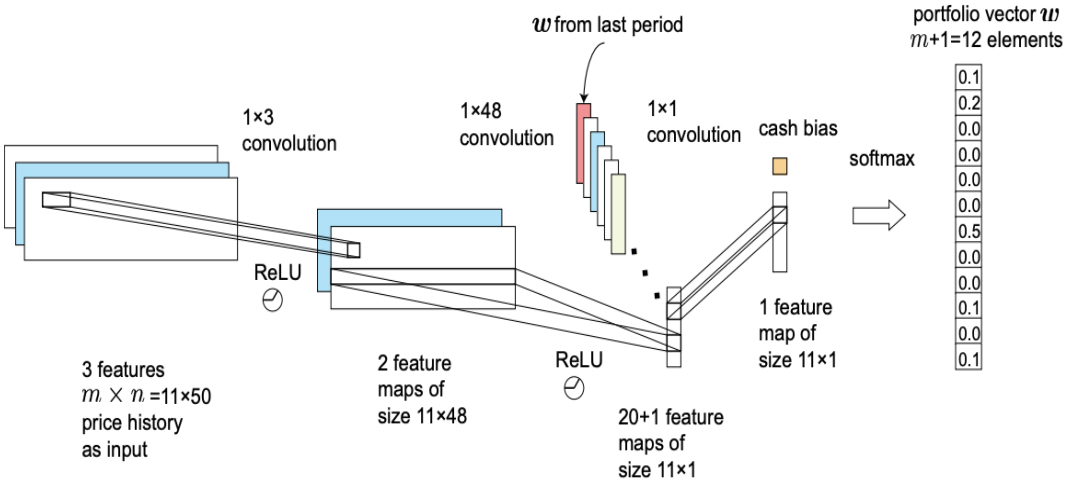


Fig: Architecture of our neural network. The input data is of shape 3 x m x n = 3x11x50 where 3 corresponds to the number of feature prices of the crypto(high, low, close), m the number of cryptocurrencies and n the number of data points. The input data is put in a sequential convolutional network. The important things here is the portfolio vector memory (PVM) which stores past experiences of portfolio return and then are reused to improve the output of next experience

Now that the algorithm is trained we will see in the next section the back-testing method used to evaluate the performance of the model.

2.4.1 Model Performance Evaluation

Before assessing the performance of our model, we proceed to cross validation to find the optimal hyper-parameters of our model. The data used ranges from '2016-05-07' and '2016-06-27'. Once down we go on to measure the performance of the model through back-testing using three data-sets. The metrics used for assessing the performance of the model are Sharp ratio(SR), maximum draw down(MDD) and final accumulative portfolio value(fAPV) which is obtained through normalisation of the accumulative portfolio value. Below is a comparison of the model performance and other preexisting performance.

	2016-09-07 to 2016-10-28			2016-12-08 to 2017-01-28			2017-03-07 to 2017-04-27		
Algorithm	MDD	fAPV	SR	MDD	fAPV	SR	MDD	fAPV	SR
CNN	0.224	**29.695**	**0.087**	0.216	**8.026**	**0.059**	0.406	31.747	0.076
bRNN	0.241	13.348	0.074	0.262	4.623	0.043	0.393	**47.148**	**0.082**
LSTM	0.280	6.692	0.053	0.319	4.073	0.038	0.487	21.173	0.060
iCNN	**0.221**	4.542	0.053	0.265	1.573	0.022	0.204	3.958	0.044
Best Stock	0.654	1.223	0.012	0.236	1.401	0.018	0.668	4.594	0.033
UCRP	0.265	0.867	-0.014	**0.185**	1.101	0.010	**0.162**	2.412	0.049
UBAH	0.324	0.821	-0.015	0.224	1.029	0.004	0.274	2.230	0.036
Anticor	0.265	0.867	-0.014	0.185	1.101	0.010	0.162	2.412	0.049
OLMAR	0.913	0.142	-0.039	0.897	0.123	-0.038	0.733	4.582	0.034
PAMR	0.997	0.003	-0.137	0.998	0.003	-0.121	0.981	0.021	-0.055
WMAMR	0.682	0.742	-0.0008	0.519	0.895	0.005	0.673	6.692	0.042
CWMR	0.999	0.001	-0.148	0.999	0.002	-0.127	0.987	0.013	-0.061
RMR	0.900	0.127	-0.043	0.929	0.090	-0.045	0.698	7.008	0.041

Fig: Back testing result of our Agent model and some preexisting trading strate-

gies. As we can clearly see, The three algorithms (CNN, bRNN, LSTM) used to model our trading Agent in overall outperform the other trading strategies in all three datasets.

Now we will tackle the second step of the report which is to modify the work done on the paper. We will based our change on the type of input data which will be data from S&P500 and on the architecture of the policy network. We will clarify all the various step leading to the final result. So let's get started.

Algorithm Modification

The method used by the original paper to tackle this optimisation problem consists in using a deep deterministic policy gradient algorithm. This algorithm is very close to the Deep Q network in that it comprises both of a actor network and a critic network, but its specificity is to use a supplementary network called target network which facilitates the reduction of the loss function.

Here we will provide a new architecture based on the previous one with some slight modification. Instead of convolutional neural networks we will use 3 CNN Given a set of assets of a portfolio of value p_i, we want to allocate optimally the weights vector $(w_i)_{1\leq i\leq n}$ across the assets so that the final return r is maximised.

3.1 Data Processing

All the work begins with the data collection and processing. Contrary to the Poleniex exchange which is open 24h/24h, we will use The S&P500 made of intraday data. The time series data is composed of stock prices characterised by 4 features Open, Close, High price, low prices. We will normalise the data to make better use of it. The input variable X_t at time t is of shape $4 \times m \times n$ where 4 corresponds to the 4 features of the price (open,high, low, close), m correspond to the number of stocks and n the number of data points. X_t can be written as :

$$X_t = [(\frac{open(t)}{open(t-1)}, \ldots, \frac{open(t-n)}{open(t-n-1)}), (\frac{high(t)}{high(t-1)}, \ldots, \frac{high(t-n)}{high(t-n-1)}), (\frac{low(t)}{low(t-1)}, \ldots, \frac{low(t-n)}{low(t-n-1)}), (\frac{close(t)}{close(t-1)}, \ldots, \frac{close(t-n)}{close(t-n-1)})]$$

3.2 Trading Environment Setup

Our goal consists in building a model to optimally allocate a vector weights $(w_t)_{i_{1\leq i\leq n}}$ to a portfolio's assets. The trading environment is defined by the set of all states S_t . The state S_t is defined by the input data at t and the set of portfolio weights allocates at t. The action of the agent correspond to the portfolio weights. Finally the reward r corresponds to the return of the portfolio. In summary we have :

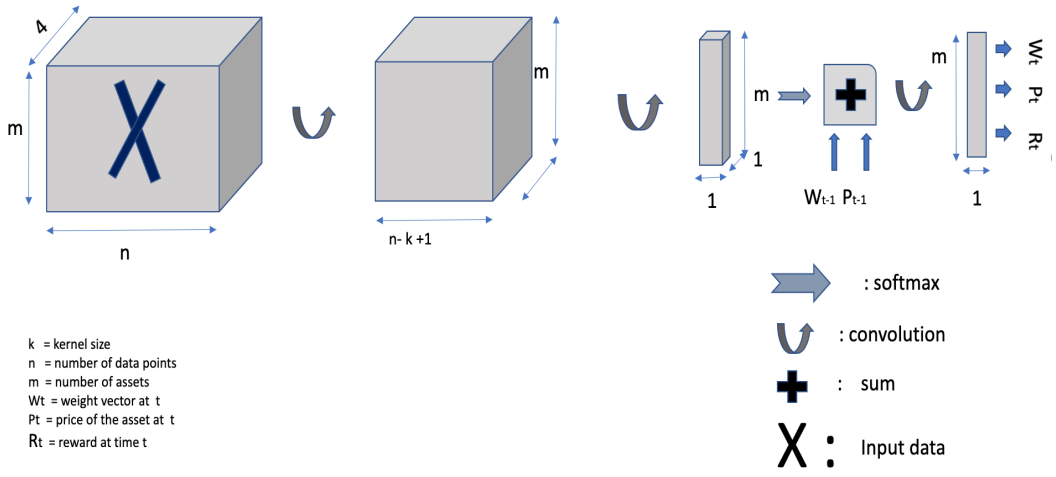
- State $S_t = (X_t, (w_t)_{i1\leq i\leq n})$
- Action = $(w_t)_{i1\leq i\leq n}$
- reward r= $\sum w_i \Delta p_i - \alpha \max(w_1, w_2, ..., w_n)$ where p_i corresponds to the return of asset i and the second term allows to not allocate all weight to one asset.

In the next section we will establish the architecture of our Policy network.

3.3 Deep Policy network

The architecture of the actor network(network used to learn the weight vector) is defined as follow:

- The input layer is a tensor of size $4 \times m \times n$
- The first convolutional layer takes this input and outputs a smaller-size tensor
- The second convolutional layer takes the output of the last layer and returns a $m \times 1 \times 1$ tensor to which is added the output from the previous time step before passing it to a softmax function.
- the last and final convolutional layer takes the $m \times 1 \times 1$ data and returns a m-vector corresponding to the action the agent should take.

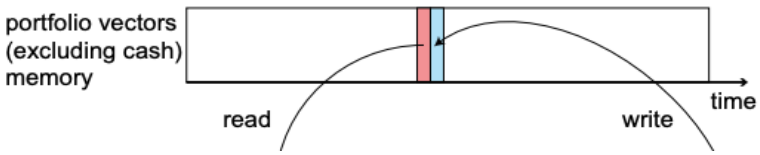


There is two main implementations of the reinforcement learning model. The first one is The PVM used to save some historical data and the last one corresponds to the Actor network. In the next section we will give a short description of these two objects.

3.4 PVM and Actor-Network classes

3.4.1 PVM Object

The Portfolio vector memory is a fixed-size vector keeping track of the weight vectors of the last time steps. The PVM vector follows the FILO(fist in last out) structure. That is whenever the vector is full, the first element added is removed so that another element can be added at the end of the vector .

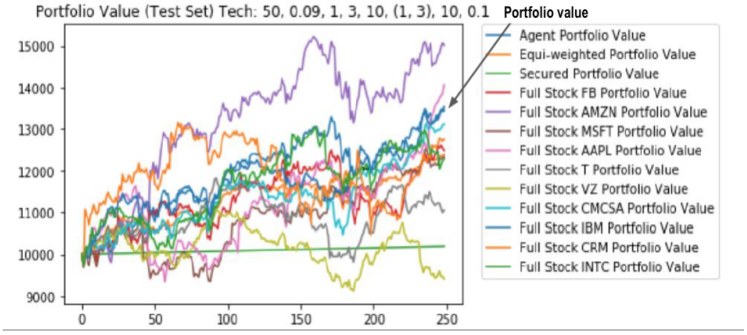


3.4.2 Actor network Object

The actor network corresponds to the network we've defined before. Here we specify the characteristic of different components of the network. For the first convolution we used a filter of size 2. For the second convolution we used a filter of size 24 . In the last layer the kernel size used for the last convolution is kernel = (1, 2).

3.5 Training and Testing of the Agent

We used the set of data range described above to train our model . After the training phase , we proceeded to back-testing. The result of the back-testing can be found in the figure below.



This figure depicts the portfolio value obtained using different strategies including that of our trading Agent. The figure clearly shows other strategies that outperform that of our agent. Contrary to the trading Agent implemented by the paper which by far outperforms all the others strategies, our model failed to do so.

3.6 Further improvement

The result showed by our model forces us to propose further improvement of our model. Maybe we could increase the number of layers and choose instead of 2 three or more layers. We only are suggesting some change that can be incorporated to the model but obviously without any implementation we wouldn't be able to conclude anything.

Conclusion

The goal of this report was to make a give a quick summary of the paper "A deep Reinforcement learning framework for for portfolio optimisation problem". This algorithm tries to solve a portfolio allocation problem which, for a given portfolio of assets, consists in dynamically allocating wealth across those assets so that the final value of the portfolio in the subsequent time is maximised. The algorithm proposed by the paper uses data from the cryptocurrency market which is open all the time.

Next we proposed our own implementation of the algorithm by using this time stock market data. The results obtained with our own-algorithm was not that conclusive because contrary to that of the original paper it does not give the optimal trading strategies. Finally, we suggest some "potential" improvement of our algorithm consisting on increasing its number of layers but any proof have been done.

The implementation of the algorithm has proven to be very difficult for us and we struggled a lot with it . We used some of the resources that were dealing with the subject online specifically the implementation proposed by the authors of the paper and implementation proposed by **Selim Amrouni** and his colleagues. We are not accustom to Reinforcement learning so the implementation task was difficult.

References

- A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem Zhengyao Jiang, Dixing Xu, Jinjun Liang
- https://github.com/ZhengyaoJiang/PGPortfolio
- https://github.com/selimamrouni/Deep-Portfolio-Management-Reinforcement-Learning