Prospects of quantum computing approach to reinforcement learning

Matteo Conterno

June 9, 2022

Abstract

Reinforcement learning is one of three major techniques that allows a model to learn, especially this technique focus on how to create an optimal agent able to reach an objective by interacting with an environment. This thesis tries to analyze the possible potentials and advantages that would derive by using quantum circuits with neural networks; analyzing and explaining how it is possible to create hybrid algorithms that exploit the advantages of both the classical and quantum algorithm, expanding the applicability not only on simulators, but even on quantum devices such as Rigetti and IonQ's processors through the amazon AWS braket service. Lastly it is analyzed how the quantum noise influence an optimal model obtained through a simulator, by testing it on a quantum processor to understand if it is possible to use the hybrid algorithm on the actual devices.

Contents

Acronyms

AI Artificial Intelligence. 5

DQN Deep Q-Network. 11

DRL Deep Reinforcement Learning. 5, 11

i.i.d. indipendent identical distributed. 10

MDP Markov Decision Process. 6, 7, 11

NN Neural Networks. 5, 9, 10

PPO Proximal Policy Optimization. 12

QC Quantum Computing. 5

QRL Quantum Reinforcement Learning. 5

SAC Soft-Actor Critic. 12

VQA Variational Quantum Algorithm. 5

1 Introduction to Deep Reinforcement Learning and Quantum Computing

This section of the thesis is created so that it can give an introduction to the fields of Artificial Intelligence(AI) specifically to Deep Reinforcement Learning(DRL) and the basis of Quantum Computing(QC) in order to understand afterward the fusion of these two different fields on Quantum Reinforcement Learning(QRL). If already expert on these subjects fell free to skip at the next section.

1.1 Approaches of learning

Currently any kind of AI requires the following components to learn:

- Data
- Model
- Approach of learning

Data is necessary in every AI model, the amount and quality can heavily influence the ability to correctly and efficiently reach his goal. The model is an algorithm that, given some data and a predefined objective, tries to complete his task using some learning approach. In order to check if the model has correctly learned, it will be later tested on unseen data and evaluated to understand if it is able to replicate the performances given a similar dataset.
The approach of learning specify how the model can learn to complete his task. At the moment there are three major ways:

- **Supervised learning** : the data is already labeled, this means that we can define when the model is incorrect or correct.
- **Unspervised learning** : the data is not labeled, this means that we can't define easily when the model is correct or incorrect and the model must uncover some pattern.
- **Reinforcement learning** : an agent interact with the environment in order to become the most optimal agent to complete the task.

This thesis will focalize mainly on the reinforcement learning approach and especially on the Deep Reinforcement Learning (DRL) which uses Neural Networks (NN) to define the most optimal agent. In this thesis the focus is on what kind of advantage can be obtained by using a quantum algorithm such as Variational Quantum Algorithm (VQA), this technique can be even used in other context and applications. Futhermore it has been demostrated that VQA and NN share some similarities and properties. The main drawback is the fact that when a VQA is trained on a classical device, there is a major overhead of time due to simulation of the quantum circuit implemented and the number of qubits is limited.

1.2 Reinforcement Learning

As said earlier a reinforcement learning approach consist of creating the most optimal agent able to complete a predefined task by interacting with the environment and taking some action. Questions arises about: how can be defined if an action is good or bad? How to model a dynamic environment? The answer is given by using a statistical model called Markov Decision Process (MDP) .

1.2.1 Markov Decision Process

In order to model a dynamic environment where an action can influence the sistem it is necessary to use the Markov Decision Process (MDP) which is an extension of Markov chains; these are a stochastic model that is able to describe a sequence of possible events that satisfy the Markov propriety, that is each event depends only on the previous event.
It is necessary to note that an MDP is based on Markov Chains that not only model the states, but even the time and this can be defined as continuous or discretized. MDP is an extension of Markov Chains because the agent can influence the state of environment and outcome, so a framework is necessary to define even his decisions and their consequences. An MDP is defined as a 4-tuple containing the following elements:

- S : set of states
- A : set of actions
- $P_a(s, s') = Pr(s_{t+1} = s'|s_t = s)$: is the transition probability of going from state s to s' by taking an action a
- $R_a(s, s')$ is the immediate reward obtained by transitioning from state s to s' by action a

The difference with a Markov chains is the presence of $P_a(s, s')$ and $R_a(s, s')$ which are necessary for the decision process, to see an example graph of and MDP see Figure 1.

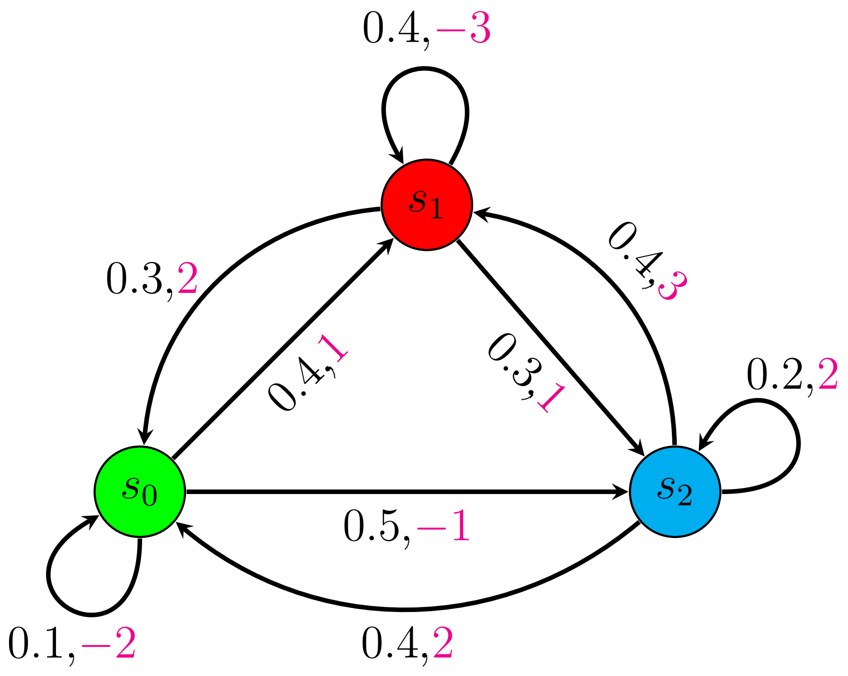

Figure 1: Markov decision process graph, transition probability is black and reward is in magenta.

The interaction between agent and environment can be defined by time, a discrete step implies to view it as distinct separate points in time uniquely definied and with a single state value can associated. The sequence of observation over times form a chain of states that is called **history**, this will be particularly important because it will be used later to define the transition probability in order to model the interaction with environment.To include the reward element of an MDP accumulated from present and future a new quantity need to be defined called **return**:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k} \tag{1}$$

The γ is a variable called discount factor with values limited inside the range of 0 and 1 with extremes included, i.e. $\gamma \in [0, 1]$. The purpose of this variable is to limit the horizon of **return**, in case γ is equal to 0 only the immediate reward will be counted and when it is 1 infinite future steps will be considered.
Usually the literature use $\gamma \in [0.9, 0.999]$ in order to gradually consider less relevant further time steps rewards thanks to the k-power of γ inside 1. The RL learning approach has the objective to maximize the **return** quantity, but this equation is not useful for the agent due to the fact that it considers every possible chain that can be observed using the Markov reward process. This means that it can vary widely even for the same state however by calculating the expectation of return from any state and averaging a large number of chains, a new quantity can be obtained called **value of state**:

$$V(s) = \mathbb{E}[G|S_t = s] = \mathbb{E}[\sum_{t=0}^{\infty} r_t \gamma^t] \tag{2}$$

These formulas considers the reward and state but they are not sufficient to model environment and agent, so it is necessary to define a set of rules to control the agent behaviour considering that the objective of RL that is to maximize the return, for these reasons a new quantity must be defined **policy**.This is formally determined as probability distribution over actions for every possible state, i.e.:

$$\pi(a|s) = P[A_t = a|S_t = s] \tag{3}$$

The policy is defined as a probability in order to introduce randomness of the agent that will be useful during the training phase, if policy is fixed the transition and reward matrixes can be reduced using policy's probabilities and decreasing the action dimension.

1.2.2 Q-learning

As stated before the objective of RL is to maximize the return, problem is how to approximate the best optimal policy and values state in order to define the correct actions for given state. Fortunately the **Bellman optimality equation** is able to estimate approximately the best action to take on deterministic and statistical case.The equation for a general case is defined as:

$$V(s) = \max_{a \in A} \mathbb{E}_{s' \sim S}[(s, a) + \gamma V(s')] = \max_{a \in A} \sum_{s' \in S} p_{a, s \to s'}(r(s, a) + \gamma V(s')) \tag{4}$$

The interpretation of this formula is that the optimal value state is equal to the action which gives the maximum possible expected immediate reward, plus the discounted long-term return of next state. This definition is recursive because the value state is determined from values of immediate reachable states. The formula not only gives the best reward that can be obtained, but it even gives the best policy to obtain that reward. Formally the policy(π) can now be defined as:

$$\pi(a|s) = \max_{a \in A} Q(s, a) \tag{5}$$

In order to simplify this formula it is possible to define other quantities, such as **value of action**:

$$Q(s,a) = \mathbb{E}_{s'\sim S}[r(s,a) + \gamma V(s')] = \sum_{s'\in S} p_{a,s\to s'}(r(s,a) + \gamma V(s')) \tag{6}$$

This quantity allows to define a pair of state and action, this is particulary important because it defines a category of learning called **Q-learning** which will be the focus of this thesis. As you can see using this new definition 4 becomes:

$$V(s) = \max_{a\in A} Q(s,a)$$

Thanks to this, the 6 can be even defined recursively and will be particulary useful later for the deep learning approach:

$$Q(s,a) = r(s,a) + \gamma \max_{a\in A} Q(s',a') \tag{7}$$

The problem is that in many situations we don't know how the value of actions, rewards, transition probabilities.Due to this it the following **Q-learning algorithm** has been created:

Algorithm 1 Q-learning

```
Require: Discount factor($\gamma \in [0,1]$)
Require: Memory table of dimension $N$ containing tuple: state($s$),action($a$), next state($s'$),
  reward($r(s,s')$) and action value $Q(s,a)$
  $i \leftarrow 0$
  for $i < N$ do
      Apply random action $a$
      Store $(s,a,s',r)$ on memory table
      Store $Q(s,a)$ with random value
      $i \leftarrow i+1$
  end for
  while Until goal is reached do
      Observe current state of sistem $s$
      Define $c(s,s')$ as counter of how many times action $a$ was taken from state $s$ to transition
      to state $s'$
      Calculate $p(s,s') = c(s,s')/\sum_{s'\in S} c(s,s')$
      Calculate $Q(s,a) = \sum_{s'\in S} p(s,s') * (r(s,s') + \gamma\, max_a Q(s',a))$
      Store $Q(s,a)$ inside memory table
      Select action $a$ from policy $\pi(s|a) = max_a Q(s,a))$
      Apply action $a$
  end while
```

This Tabular Q-learning present major drawbacks such as:

- A large memory table is required to store all the values used to approximate the $Q(s,a)$ values that will be used for the $\pi(s|a)$
- Complete iteration over all possible states is required in order to extract the values that will be stored inside the memory table

In order to resolve these drawbacks a new type of Q-learning algorithm called **Tabular Q-learning** was invented. The main difference is the lack of necessity to iterate over all the states to optimize because only the ones obtained from the environment will be used for optimization. Futhermore the table will only contain the $Q(s,a)$, this does not unfortunately resolve completely the drawback to have a large memory table, but at least reduce it.

Algorithm 2 Tabular Q-learning

```
Require: Discount factor $\gamma \in [0, 1]$
Require: Learning rate $\alpha \in [0, 1]$
Require: Memory table of dimension $N$ containing action value $Q(s, a)$
  loop
    Create a table with initial values for $Q(s, a)$
    Select random action $a$
    Observe the tuple $(s, a, r, s')$
    Calculate $V(s') = \max_{a' \in A} Q(s', a')$
    Update $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma V(s'))$
    Store $Q(s, a)$ inside table
    Test episode using $\pi(a|s) = \max_{a \in A} Q(s, a)$ with $Q(s, a)$ of stored values
    if goal is reached then
      break loop
    end if
  end loop
```

As noted, a memory table is required but only to store the $Q(s, a)$ values and it will be used to update itself and to define which action need to be taken following the policy. There is still one major problem though, this algorithm will struggle if there is a large count of the observables state set. This, in many real life situation, is quite common for example on the Cartpole environment of OpenAI gym that we are going to use there are only 4 states, but the interval of values that can be taken is enormous. A solution of this problem would be to use a non-linear representation of action and state into a value. This is a typical "regression problem" in the field of machine learning and thanks to the power of Neural Networks (NN) it is possible to approximate any kind of linear or non-linear function given enough data. Fortunately the amount of data is almost limitless thanks to the fact that if new data is needed, the only requirement is to interact more with the given environment.So now, it will be introduced the **Deep Q-learning** algorithm.

1.3 Deep reinforcement learning

Deep reinforcement learning is an extension of the classic one due to the use of Deep Learning techniques such as the Neural Networks (NN). Neural Networks has been introduce by [1] who largely inspired by the biological brain with neurons and connections. At that time many limitations were observed due to the single layer architecture and small amount of neurons that the hardware was able to simulate, only thanks to the backpropagation algorithm introduced in [2] it was later possible to train neural networks with more layers and neurons. Furthermore as demonstrated by [3] a neural network with only one single hidden layer is able to approximate any kind of linear or non-linear function, but the paper doesn't tell how many neurons are required. Even if the algorithms were introduced in the last century only in the last two decades thanks to hardware advancements such as GPUs and large amount of datas from the first databases it was possible to train effeciently these neural networks. Fortunately reinforcement learning can increase the amount of data by increasing the time of interaction with the environment, but the question on how many neurons a neural networks requires to approximate doesn't have a solution, but only indications. Now that the neural networks have been briefly introduced it is time to introduce the variation of Tabular Q-learning(1.2.2) that uses neural networks to solve the environment.

1.3.1 Deep Q-learning

Tabular Q-learning is able to solve the problem of iteration over time, but it still struggles with situations when the count of observable state set is very large. This is a typical situation in real life where the set of states can be even infinite, solutions have been proposed such as use bins to discretize, but in many cases it did not result successful. A better solution is to create a nonlinear representation, instead of a linear one, that maps both state and action to a value. This is commonly called in machine and deep learning as "regression problem" and in general does not require neural networks, but this approach is the most popular one. Generally the following point are required to train a neural network to solve an environment:

1. Initialize $Q(s,a)$ with some initial approximation.
2. Interact with the environment to obtain tuple (s,a,r,s').
3. Calculate loss, if episode ended is $\mathcal{L} = (Q(s,a) - r)^2$ otherwise is $\mathcal{L} = (Q(s,a) - (r + \gamma \max_{a' \in A} Q(s',a')))^2$. The loss can be defined in other ways but it needs to have a difference between value of actions from current state and the discounted reward or only reward if episode ended.
4. Update $Q(s,a)$ using an optimizer algorithm to minimize loss.
5. Repeat from step 2 until convergence or goal reached.

Even if it looks simple, some modifications are required in other to ensure convergence. First of all when it is necessary to store the experience we need to find a strategy to explore the environment and optimize the current approximation, this is often referred to "exploration versus exploitation". In order to have an efficient solution it is good to behave randomly at the beginning because Q-value approximation is bad at that time and later start to act using the Q-value obtained. Usually it is used the **epsilon-greedy method** where a random probability value is sampled from a uniform distribution and confronted with a probability ϵ that tells the algorithm to behave randomly. ϵ is usually initialized to 1 so that any kind of random value sampled is smaller to ϵ so that the model behave randomly and later ϵ is decreased to a final small value so that the model will not always behave randomly but use the Q-values approximated. After that this problem is solved, there is another matter linked to how an optimizer algorithm work on a NN. The gradients calculated from input data requires that the data are i.i.d., otherwise there would be incorrect estimations of the correct direction on which the parameters must be updated. Furthermore the data must not be completely randomic but must reflect the past actions taken in order to have an experience that tells which action are worst than others. To solve this, it is necessary to create a **large memory buffer** in order to store the past actions taken both randomic and by exploiting the agent. Last problem that needs to be addressed is due to the steps correlation, when we calculate the loss we use $Q(s,a)$ and $Q(s',a')$ using the same NN, this can be a problem because s and s' are highly correlated and this can lead to similar approximation. This influence the convergence of learning during the training process, to deal with it a copy of the NN is created and updated by copying parameters of the original one with a fixed interval, this NN is usually called *target network.* The following pseudo is taken from the original paper that was tested on atari games [4], after this paper numerous variations have been proposed in order to improve convergence and reduce time of training, such as Double Q-Net [5] and many others. Due to the limited resources and time required the thesis will be focused on the "classical" DQN, but it would be interesting to verify if the advantage would be more pronounced or reduced.

The 1.3.1 gave a breakthrough on this field thanks to the fact that was able to achieve a human-like, and in some cases even better, performance on multiple games of the atari.

Algorithm 3 Deep Q-Networks

```
Require: Memory buffer of dimension $N$ containing tuple: state($s$),action($a$), next state($s'$),
  reward($r(s, s')$)
Require: Discount factor $\gamma \in [0, 1]$
Require: Learning rate $\alpha \in [0, 1]$
Require: Optimizer
Require: A neural networks for $Q$ and target network for $\hat{Q}$
Require: $\epsilon$ probability of randomness
  Initialize $Q(s, a)$ and $\hat{Q}(s, a)$ with random weights and empty memory buffer, $\epsilon \leftarrow 1.0$
  while goal not reached do
    Observe state and define with probability $\epsilon$ if $a$ is random or $a = \arg\max_{a \in A} Q(s, a)$
    Apply $a$, observe $r$ and $s'$, store in memory buffer the tuple $(s, a, r, s')$
    Sample a batch of tuple $(s, a, r, s')$
    For every tuple calculate $y = r$ if episode ended for that state, otherwise calculate $y =$
    $r + \gamma \max_{a' \in A} \hat{Q}(s', a')$
    Calculate loss $\mathcal{L} = (Q(s, a) - y)^2$
    Use optimizer to update $Q(s, a)$ parameters in order to minimize $\mathcal{L}$
    Every $n$ steps copy weights of $Q$ to $\hat{Q}$
  end while
```

Thanks to this Deep Reinforcement Learning has been tested and applied in other contexts such as finance, medicine, robotics and many others. Before we introduce quantum computing it is necessary to explain and describe another algorithm that is able to handle Markov Decision Process without the value iteration method: **policy gradient methods**.

1.3.2 Policy gradient methods

The DQN that we have illustrated is focusing mainly on approximating the value of actions(Q) and the value of state(V) so that afterward the choice on which action to take is based on a greedy approach: the best action to take is the one that maximize the return. This is not incorrect, but in some cases it may not be good due to the environment consequences. For this reason in those cases it is better to focus on how to define the agent behaviour in order to consider vene the possible alternatives. Another reason why these methods are used is due to the fact that can introduce **stochasticity** and can work on **continous environment** differently from the DQN showed. Now that the method is focused on policy, it is necessary to give a **policy representation** in order to work with it, the most common way is to use a probability distribution over the possible actions. This gives an additional advantage to neural network which is to have a smooth representation, in other words if a variation is applied on the weights outputs variate. It is now time for the final step, to find a way to optimize the weights in order to improve the policy. Thanks to the policy gradient theorem, it is known that the formula we need to use is:

$$\nabla J \approx \mathbb{E}[Q(s, a) \nabla \log \pi(a|s)] \tag{8}$$

the complete demonstration of the formula and application on Deep Reinforcement Learning can be found on [6]. The meaning of this expression can be as follow: the policy gradient tells us the direction of update, the formula is proportional to the value of action taken $Q(s, a)$ and the gradient of log probability action taken. Simply we are trying to increase the probabilty of good actions and rewards, at the same time decreasing the probability of bad actions and outcomes. Finally, the $\mathbb{E}$ is expectation value due to the fact that will be used with multiple values. There are some drawbacks for this approach such as a high gradients variance that can influence the learning and exploration at the beginning of training and to avoid falling in a local minima it is necessary to introduce some kind of uncertainty or *entropy.* There is another problem of policy gradient methods and that is the fact that are usually less sample-efficient and this implies a bigger numeber of iterations to solve the environment. In order to tackle this problems new algorithms have been defined, notably Actor-Critic (A2C) [[7]], Proximal Policy Optimization[8] (PPO) and others which tries to reap the benefits of policy and value methods. This thesis will use the Soft-Actor Critic (SAC) algorithm on a robotic environment and later will confront it with the quantum type.

1.3.3 Soft Actor Critic

This algorithm can be seen as a variation of Actor-Critic, the latter one can be simplified as the necessity to improve the behaviour and values of states approximated in order to solve the environment. In order to achieve this two neural networks are used: one to approximate the policy called **policy net or actor** and one to approximate the value of state called **value net or critic**. The problems of this algorithm is the fact that is on-policy and sensible to

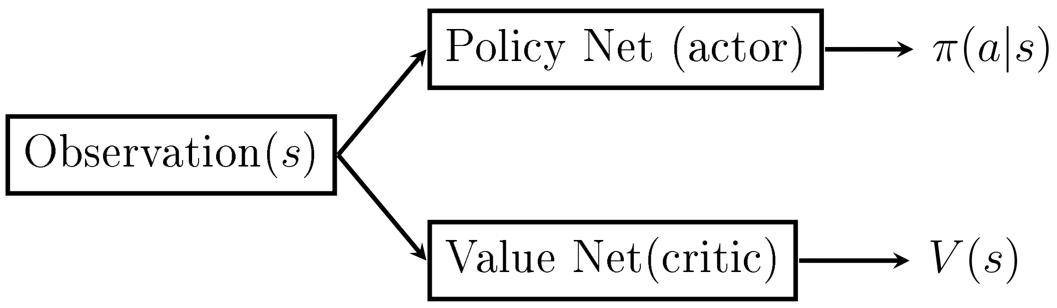

Figure 2: Schematic on actor-critic architeture

hyperparameters, in order to improve performance, stability and efficiency the SAC was created. SAC is able to do it introducing multiple components such as :

- a large buffer memory is needed to efficiently update the weights using previous history increasing in this way stability and switching to an off-policy method instead of an on-policy one.
- A target network of the critic net is introduced to improve the stability and efficiency of learning to approximate the value state, differently from the DQN updating target network will follow the procedure of **soft functions update**. Shortly, this means that the values will use a factor to combine new and old weights obtained from the update step, to do this a loss value of state (J_V) and action (J_Q) function must be defined.

$$J_V(\psi) = \mathbb{E}_{s \sim S}\left[\frac{1}{2}(V_\psi(s_t) - \mathbb{E}_{a_t \sim \pi_\phi}[Q_\theta(s_t, a_t) - \log \pi_\phi(a_t|s_t)])^2\right] \tag{9}$$

$$J_Q(\theta) = \mathbb{E}_{s \sim S, a \sim A}\left[\frac{1}{2}(Q_\theta(s_t, a_t) - r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim p}[V_{\overline{\psi}}(s_{t+1})])^2\right] \tag{10}$$

- the maximum entropy of reinforcement learning is introduced to increase the exploration of environment, improve learning and reduce sensibility to hyperparameters.The loss of this maximum entropy was transformed for this case from the "classical" one considering the use of a neural network and introducing an input noise vector(ϵ_t) sampled from the a fixed spherical gaussian distribution to calculate the action.

Output action of neural network and input noise from currrent state can be expressed as follows $a_t = f_\phi(\epsilon_t; s_t)$ bringing to the loss formula:

$$J_\pi(\phi) = \mathbb{E}_{s\sim S,\epsilon_t\sim\mathcal{N}}\left[\log \pi_\phi(f_\phi(\epsilon_t; s_t)) - Q_\theta(s_t, f_\phi(\epsilon_t; s_t))\right] \tag{11}$$

For more details on how these formulas have been obtained and experimental results refer to the original paper [9], for simplicity only the pseudocode will be showed considering that there will be slight modifications in the actual code used to correctly confront classical and quantum.

1.4 Quantum computing: an introduction

Bibliography

[1] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65 6 (1958), pp. 386–408.

[2] Ronald J. Williams David E. Rumelhart Geoffrey E. Hinton. “Learning representations by back-propagating errors”. In: *Nature* 323 (1986), pp. 533–536. DOI: `https://doi.org/10.1038/323533a0`.

[3] Halbert White Kurt Hornik Maxwell Stinchcombe. “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 5 2 (1989), pp. 359–366. DOI: `https://doi.org/10.1016/0893-6080(89)90020-8`.

[4] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: *CoRR* abs/1312.5602 (2013). arXiv: `1312.5602`. URL: `http://arxiv.org/abs/1312.5602`.

[5] 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`.

[6] Richard S. Sutton et al. “Policy Gradient Methods for Reinforcement Learning with Function Approximation”. In: NIPS’99 (1999), pp. 1057–1063.

[7] Ziyu Wang et al. “Sample Efficient Actor-Critic with Experience Replay”. In: *CoRR* abs/1611.01224 (2016). arXiv: `1611.01224`. URL: `http://arxiv.org/abs/1611.01224`.

[8] John Schulman et al. “Proximal Policy Optimization Algorithms”. In: *CoRR* abs/1707.06347 (2017). arXiv: `1707.06347`. URL: `http://arxiv.org/abs/1707.06347`.

[9] Tuomas Haarnoja et al. “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor”. In: *CoRR* abs/1801.01290 (2018). arXiv: `1801.01290`. URL: `http://arxiv.org/abs/1801.01290`.