UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

OFFLINE REINFORCEMENT LEARNING

Diploma Thesis

VASILOPOULOS VASILEIOS

Supervisor: Tsalapata Hariklia

Volos year 2021

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

OFFLINE REINFORCEMENT LEARNING

Diploma Thesis

VASILOPOULOS VASILEIOS

Supervisor: Tsalapata Hariklia

Volos year 2021

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΚΤΟΣ ΣΥΝΔΕΣΗΣ ΕΝΙΣΧΥΤΙΚΗ ΜΑΘΗΣΗ

Διπλωματική Εργασία

ΒΑΣΙΛΟΠΟΥΛΟΣ ΒΑΣΙΛΕΙΟΣ

Επιβλέπων: Τσαλαπάτα Χαρίκλεια

Βόλος 2021

Approved by the examination Committee:

Supervisor	**Tsalapata Hariklia** E.DI.P, Department of Electrical and Computer Engineering, University of Thessaly
Member	**Daskalopoulou Aspasia** Assistant Professor, Department of Electrical and Computer Engineering, University of Thessaly
Member	**Vassilakopoulos Michael** Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Date of approval: 22-02-2021

ACKNOWLEDGEMENTS

Ευχαριστώ όλους όσους βοήθησαν στην περάτωση της διπλωματικής.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

ABSTRACT

With the current popularity of neural networks, both supervised and unsupervised learning have received a tremendous amount of recognition and have been used everywhere. Reinforcement learning had been on the background for quite a while , just because it required a breakthrough , which happened with the introduction of Deep Reinforcement Learning. DeepMind practically revived deep RL , a technology which dates back to the 80's and was implemented by TD-Gammon[16] (one of the first successful RL applications using neural networks). As a result, RL's popularity began to rise and found its usage in many games, aswell as robotics, autonomous driving, healthcare. RL , in contrast with supervised learning which is trained by a dataset, is a reward based method, where the agent chooses actions that yield rewards, either positive or negative, that help him train. In this particular thesis, we will explore the world of RL , the particular reasons why it holds such great promise and focus on offline RL , which is a form of RL that uses supervision in order to learn. There will also be a reference to neural networks and deep learning, as an important piece in the promotion of reinforcement learning. We will also provide currently open topics for research, as well as datasets that are currently free to use and are being offered in order to promote the field of offline reinforcement learning.

ΠΕΡΙΛΗΨΗ

Με την τρέχουσα δημοτικότητα των νευρωνικών δικτύων, η επιβλεπόμενη καθώς και η μη επιβλεπόμενη μάθηση , έχουν λάβει ένα σημαντικό ποσό αναγνώρισης και ουσιαστικά χρησιμοποιούνται παντού. Αντιθέτως, η ενισχυτική μάθηση κειτόταν στο βάθος της επικαιρότητας, αφού χρειαζόταν μία εξέλιξη, η οποία κατέστη δυνατή με την άνοδο της βαθιάς ενισχυτικής μάθησης. Η εταιρεία DeepMind έφερε στο προσκήνιο την βαθιά ενισχυτική μάθηση , η οποία χρονολογείται στη δεκαετία του ’80 και εφαρμόστηκε το 1992 στο TD-Gammon (μία από τις πρώτες επιτυχημένες εφαρμογές ενισχυτικής μάθησης χρησιμοποιώντας νευρωνικά δίκτυα). Αυτό είχε ως αποτέλεσμα, η βαθιά ενισχυτική μάθηση να χρησιμοποιηθεί σε τομείς όπως παιχνίδια , ρομποτική, αυτόνομη οδήγηση και ιατροφαρμακευτική περίθαλψη. Η ενισχυτική μάθηση ,σε αντίθεση με την επιβλεπόμενη που χρησιμοποιεί ένα σύνολο από δεδομένα για να εκπαιδευτεί, έχει τη μορφή της μάθησης μέσω ανταμοιβής , όπου ο πράκτορας επιλέγει ενέργειες οι οποίες του δίνουν ορισμένες ανταμοιβές , είτε θετικές είτε αρνητικές, μέσω των οποίων εκπαιδεύεται. Στη συγκεκριμένη διπλωματική, θα εξερευνήσουμε τον κόσμο της ενισχυτικής μάθησης, τους λόγους για τους οποίους είναι πολλά υποσχόμενη και θα αφοσιωθούμε στην εκτός σύνδεσης ενισχυτική μάθηση, η οποία είναι ένα είδος ενισχυτικής μάθησης που χρησιμοποιεί επίβλεψη για να εκπαιδευτεί. Θα γίνει επίσης αναφορά σε νευρωνικά δίκτυα, καθώς αποτελούν ένα σημαντικό κομμάτι της ανάδειξης της ενισχυτικής μάθησης. Τέλος , θα αναφέρουμε και ανοιχτά ερευνητικά θέματα ,καθώς και δεδομένα τα οποία διατίθενται δωρεάν από ερευνητικά κέντρα για την ανάδειξη του χώρου της εκτός σύνδεσης ενισχυτικής μάθησης.

TABLE OF CONTENTS

List of Figures

CHAPTER 1

INTRODUCTION

RL is an area of machine learning that possesses a wide variety of innovations and improvements over the years. The exact steps that were undertaken in order to lead us to today's state of RL are referenced in [3] and split into two distinct categories.

The first breakthrough became apparent around the mid-50's when Richard Bellman introduced the notion of "optimal control"[17] based on the works of Hamilton and Jacobi a century earlier [3]. This formed the basis for Dynamic Programming, the Bellman Equation that gave birth to all the elements contained in the field of RL. However, control theory implementations revolve around the idea that the environment in a given setting is perfectly known, which is somewhat counterintuitive to the whole aspect of RL being trial and error. But when and how were the foundations set on RL to intuitively become acknowledged as trial and error?

The works of [17][18] gave birth to a very prominent methodology, namely the Temporal Difference Learning. There were also projects that date earlier, which introduced trial and error setting but it was these papers that set the ground for the current state of RL. TD methods originate from studies on animal learning and were conjuncted by the works of Harry Klopf [19][20] .Both the TD methods and optimal control problems were unified under the scope of Q-learning[21], which is basically a TD control algorithm described later in Chapter 3.

In this thesis our aim is to provide the reader some insight on the very broad topic of RL and focus on a specialized field called Offline Reinforcement Learning, a subset of RL. This of course, cannot be achieved without a reference to machine learning and neural networks.

So, in Chapter 2 we provide the reader with some basic information about machine learning and different kinds of neural networks that are consistently used in RL and will assist at understanding the algorithms later on. Also, grasping fundamental knowledge about different kinds of learning, such as supervised or unsupervised, will definitely prove to be helpful in understanding the importance of RL.

In Chapter 3, we familiarize the reader with RL theory and algorithms and some of the most important methodologies currently being in use in the field. The comprehension of such notions is obligatory for the offline RL field to become understandable.

In Chapter 4, we present the most recent advancements on the topic of offline RL and why it is a key factor in the development of a wide range of recent technologies. Furthermore, the results from different experiments conducted by authors of several papers involving offline RL are summarized and the conclusions that are being deduced are displayed, noting the significance of the field.

In Chapter 5, we take a holistic approach to RL as a whole, providing insight on how offline RL can aid in the evolvement of the field, discussing the material presented in the thesis and referring to open problems currently open for research, as well as datasets that can provide useful to a researcher.

CHAPTER 2

DEEP LEARNING

2.1 Introduction to Machine Learning

When we hear of the term Machine Learning we probably think of machines learning using some kind of technique. Machine Learning is actually that and is comprised of 3 particular main types:

1. Supervised Learning: A class of problems that involve learning a mapping from inputs to outputs, with them belonging in a dataset. There are 2 types of supervised learning, classification which involves the prediction of a class and regression which involves the prediction of a numerical value.
2. Unsupervised Learning: A class of problems whose outputs are unknown and the algorithm tries to make sense of the input data.
3. Reinforcement Learning: A class of problems that involve an agent acting on the environment and getting the maximum possible cumulative reward from all these actions.

So, supervised learning is basically approximating the function g with input variables X and Y where $Y = g(X)$. An important concept that arises from that is the need to generalize to new data that the model has not seen. The term generalization refers to how well the model responds to new input data presented to it, having learnt from the training data we provided. Two issues arise as problems of generalization, overfitting and underfitting.

Overfitting is apparent when the model learns from the data too well, meaning that it models the noise of the training data as a part of them. Overfitting is usually present in the case of nonlinear models where the model is much more flexible to exactly fit the data than linear models who don’t possess the flexibility to completely match the training examples. On the other hand, underfitting refers to a model not being able to either generalize or model the training data. We do not usually care about this case, because it’s easy to detect, unlike overfitting. A solution that’s used for overfitting is employing a validation set, which is used after the model has seen the training data to evaluate how well the model reacts to previously unseen data.

Unsupervised learning is , as we saw , trying to label unlabeled data into clusters. The data is being handed to the model and it tries to correlate them with an outcome, with the goal being the identification of hidden patterns in the data that the computer will unearth. The main methods in which we apply unsupervised learning are clustering, that is forming clusters of data and anomaly detection, which is discovering extreme cases in the data that are usually denoted as suspicious, something that's useful in malware and fraud detection. It is a generally complex type of learning which we will not get too much into in this thesis.

Reinforcement learning is a subset of ML that allows agents to experiment with their environment and learn from it. It has had increasing success, as it resembles the learning of human beings and animals in general, a process that seems very natural. The model can learn from experience and is useful for achieving long term goals, something which is hard to accomplish. It suffers from a lot of problems aswell , with some being the curse of dimensionality, a problem that comes up when we analyze high-dimensional spaces, the hunger for computation resources as the model needs to replay the situation especially in video games and the assumption that the world is Markovian (explained at Chapter 3). But one of the biggest disadvantages and the one we will talk about the most in this thesis is real-world sampling, a problem which rises from the usage of RL in robotics and autonomous driving, where we don't have the luxury to perform replays as the hardware is usually too expensive. That can be combated with offline RL , which we will address in Chapter 4.

The rise in popularity of these machine learning categories is definitely correlated with the advance of deep neural networks - that was caused by the advancement of GPUs-, just because it is now possible to model high-dimensional input data. As a consequence, deep neural networks are an integral part of machine learning and have helped the technology to rise steadily over the years.

2.2 Feed-Forward Neural Networks

Artificial neural networks are function approximators that were inspired by biological neural networks. They consist of the input and output layers aswell as hidden layers that exist between them. Their role is to handle input data, adapt to them changing and estimating the output by performing calculations in the hidden layer(s). Specifically, Feed-Forward neural networks are the simplest type of neural networks. Their main feature is that the movement is only forward , from the inputs to the outputs, so there are no loops or cycles involved.

2.2.1 Perceptron

The unit and simplest form of a feed-forward neural network, is the perceptron. As we can observe in the image below, the perceptron performs the cartesian product of the inputs and the weights, which is called the weighted sum. The activation function, which we can arbitrarily choose but has an impact on the convergence and the general behavior of the neural network, provides a mapping from the weighted sum to the values of the function. So the result that stems from the activation function is the output and the output decides the weight updates. That means that the weights are updated constantly until all outputs are classified correctly. In general, perceptrons are guaranteed to converge when the data they are provided with are linearly separable but for nonlinear approximation we perform different strategies.

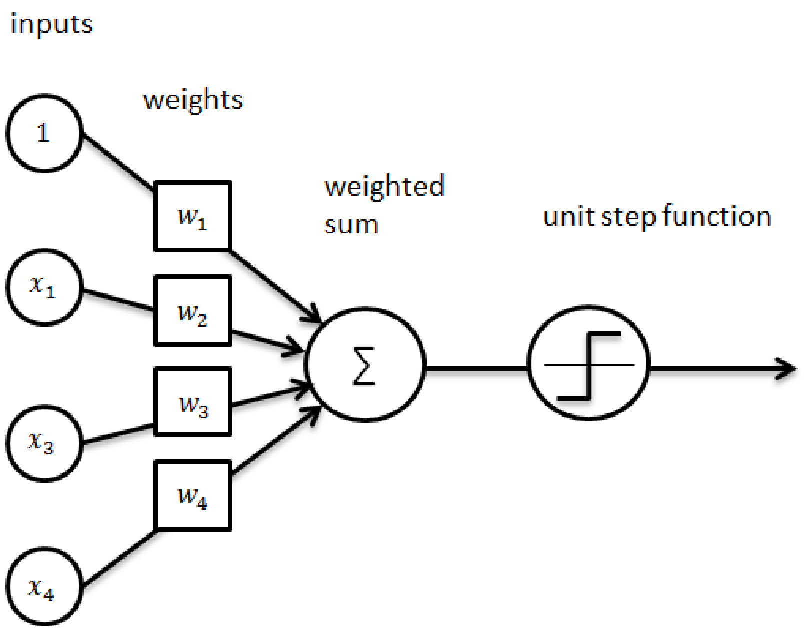


Figure 2.1: Representation of a perceptron[41]

Let’s analyze the components of a perceptron:

- Weights : They define the direction of our decision hyperplane
- Net input function : The sum of the weighted inputs that gets fed to our activation function
- Activation Function : An activation function is a function that handles the sum of the weighted input data. The output of a perceptron results from this interaction. Three types of activation functions are mainly used and are the following:
    - Ridge functions: Functions that are a linear combination of inputs (step function, logistic function)
    - Radial functions: Functions whose value at each point depends on their distance from the origin and that point (Gaussian function)
    - Folding functions: Functions that are mainly used in Convolutional NN’s and classification, because they are used to calculate things like the mean, maximum, minimum of a set of inputs.

Each activation function has its own usage, meaning that usually we pick an activation function, based on the specific function that we are trying to approximate. For instance, if we encounter a classification problem our first guess is to choose the sigmoid function, as it is a great depiction of what a classifier should resemble.

- Error : The error is the difference between the actual output and the desired output.
- Perceptron learning rule : The weights of a perceptron are updated after each step, if our training output, fails to match our target output, according to the learning rule, also called delta rule[38]:

  $w_i = w_i + \Delta w_i,$

  where Δw_i is equal to

  $$\Delta w_i = a(t - o)x_i$$

  where a is the learning rate (we prefer to keep it low in order not to blow up the weights) , t is the target value (that’s provided in our dataset) , o is our predicted output, and x_i is the i'th input. This process ends when every pattern matches the target value and the linear function is approximated.

We just analyzed the simplest form of an ANN but the truth is that we are kind of limited to linear models because non-linear approximation is out of the question using plain perceptrons. A solution to this is Multi Layer Perceptron.

2.2.2 Multi Layer Perceptron

MLP is a deep neural network that consists of perceptrons. More specifically, it contains an input layer that receives the input data, an output layer which classifies or predicts the input and between those, any number of hidden layers. MLPs are fully connected , meaning that each node in a layer is connected with all the nodes in the next layer, until we reach the output node. The thing that's new in this network is the learning update, the notorious backpropagation which is a generalization of the previous seen delta rule, used in perceptrons.

Backpropagation is a backward pass that occurs after the forward pass of the network. After the error is calculated, we need to update the weights but that is done by checking the 'sensitivity' of them related to our output. So we basically apply the chain rule in order to calculate a gradient of the form $\frac{dOutput}{dWeight}$. After we compute the gradient then we perform our gradient descent algorithm to update the weight.

$$weight = weight - a * \frac{dOutput}{dWeight}$$

The whole process of backpropagation is to calculate the partial derivatives from the error function to the neuron with the specific weight that we want to update. When all of the weights of the network are updated, then we can begin the forward pass again.

2.2.3 Convolutional Neural Network

CNNs[28][11] are a type of feed-forward neural network that is being used extensively in RL and is mainly used in case of images , i.e two-dimensional data. CNNs are composed of convolution layers and a fully connected layer at the end that classifies the input. The process of training in these networks, as we see in Figure 2.2, comprises of x steps.

First, in every convolution layer a convolution is performed with our input data and the filter that we're using and an output,which is called feature map, is extracted. The feature map is passed to a pooling layer that downsamples the dimensionality of it. One of the most popular ways of achieving this is max pooling, a method that extracts patches from our feature map and outputs the maximum of each patch. For example, a max pooling with a 2x2 filter and a stride(the amount that the filter is shifting) of 2, outputs one element (the maximum) for every 2x2 array in our feature map. When we reach the last convolution or pooling layer and receive the final feature map, we transform it to a 1D array (flattening) and connect it to a set of fully connected layers that map to the outputs of the network. The final fully connected layer usually has the same number of output nodes as the number of classes.

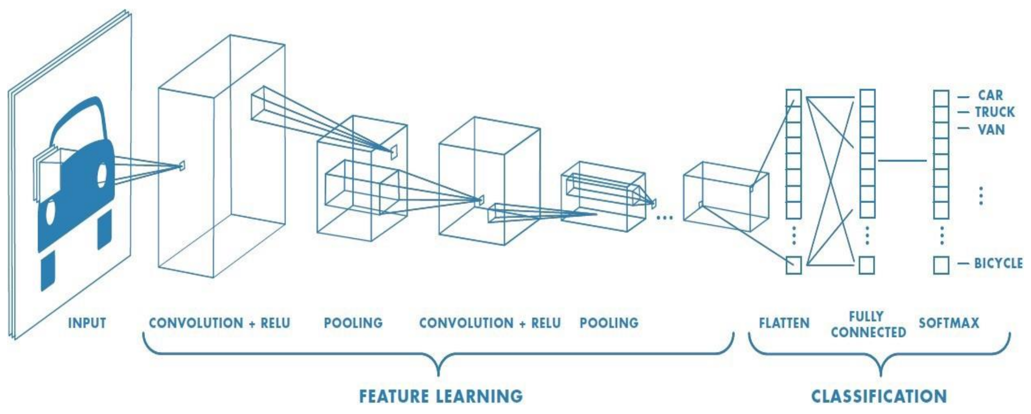


Figure 2.2 : A convolutional neural network[42]

The advantage of using CNNs over MLPs in image classification is that a MLP requires great loads of hardware in order to train high resolution image with all the connections that are required. CNNs on the other hand, take advantage of the temporal and spatial features of an image, meaning that if we see a cat in an image , it will not matter whether the cat is on the top right corner or the top left corner, whereas with MLPs we would have to relearn the same feature extractor for the different locations. With all that said, CNNs are an obvious choice for this task.

2.3 Recurrent Neural Networks

RNNs are a type of neural network that differ from feedforward NNs in the sense that they can loop backwards and not just follow a straight path. They are mainly used for sequential data, I.e data that follow a pattern , because they store memory of past actions. So, the architecture is having loops in the hidden layers that feedback the previous inputs to our node, along with our current input. That means that the input that is fed to the NN is comprised of current and past inputs, so it can model the whole sequence and produce more reliable results regarding that sequence.

A RNN updates its values using some form of backpropagation called backpropagation through time, which basically is backpropagation taking into acoount multiple timesteps. BPTT unrolls all input timesteps and each timestep is regarded as a layer of a network. As we can guess, if a network has loads of timesteps then the backpropagation may cause weights to approximate zero or overflow. An activation function like hyperbolic tangent that has values in the range(-1,1) may cause the first case , while gradients with a value above 1, when accumulated, can cause the second case.

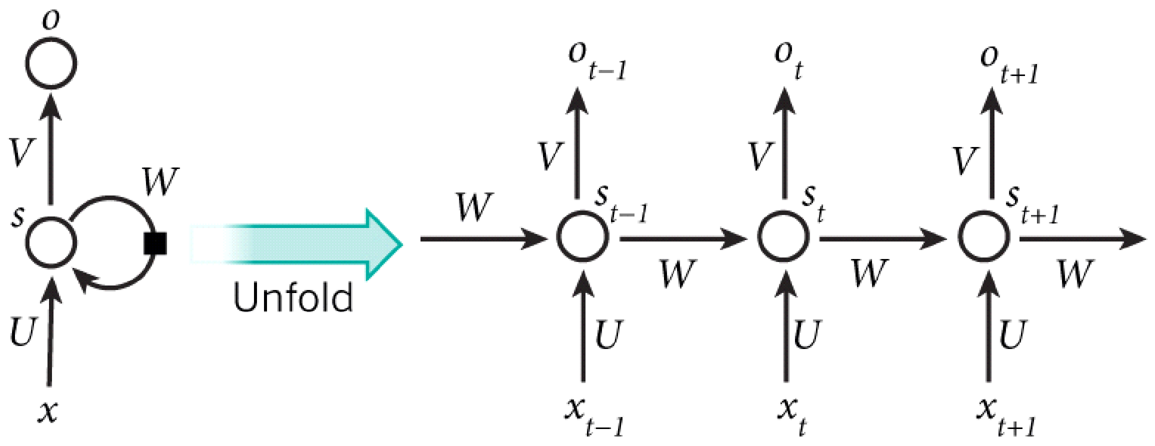


Figure 2.3 : A recurrent neural network and its unfolding in time[43].

X_t is the input at timestep t

S_t is the hidden state at timestep t and is calculated using the previous hidden state and X_t by $f\left(Ws_{\{t-1\}} + \ Ux_t\right)$ where f is an activation function like tanh or ReLU

O_t is the output at timestep t

As we can observe, S_t holds information about what happened in previous timesteps but in practice it's hard to retain information about states that occurred a lot of timesteps ago. That is improved with the introduction of LSTMs.

LSTMs are used in order to tackle the problem of long-term dependencies, i.e the problem of remembering long sequences and the problem of vanishing gradient. Their characteristic is that in each timestep , the repeating module has 4 layers instead of 1 in the classic RNNs, which form the forget gate, input gate and output gate. That way we are able to keep the useful information that we gathered from previous data. There are other variants of LSTM networks , most notably GRU , that are gaining popularity.

2.4 Radial-Basis function Networks

Radial-basis function networks resemble feed-forward neural nets but differ on the activation function which is non-linear. They usually contain 3 layers , the input , the hidden and the output layer. They differ from MLPs , because instead of just multiplying the inputs with the weights and summing the results in the end, they compare the inputs with the trained values producing a similarity value, which is then multiplied with the weights and summed in the output. The most common function used in RBFs is the Gaussian that takes into account the input and the distance of the input from the center.

2.5 Variational Autoencoders

A technology that's helpful in many situations where supervised learning does not apply, are variational autoencoders. They work with unlabeled data, that is they belong in the unsupervised learning region, but contrary to other famous unsupervised algorithms like k-means, they are practically neural networks. They are composed of two main parts, the first one being the encoder and the second one being the decoder. The first part is about receiving an input and encoding it into the latent space and the second part is about recovering that input and outputting it. The whole idea of this method is to figure out the underlying connection between features, as VAs are generative models that try to learn a probability distribution instead of discriminative models that are aimed towards mapping the input data into a prediction[9].

In mathematical terms, the encoder receives an n-dimensional input x, which for example might be an image with $\sqrt{n} * \sqrt{n}$ pixels and maps them to a latent space z ,with it being a probability distribution and having less dimensions than the encoder. So the encoder is represented as $q_{\varphi}(z|x)$. Then using the information from z as an input, the decoder outputs x which is now an n-dimensional space like before, that contains a value between 0 and 1 for each element, in our example for each pixel.

CHAPTER 3

REINFORCEMENT LEARNING

3.1 Introduction

We've briefly discussed about RL in the previous chapters but now we're going to dive deeper into the matter. In this particular subsection of the chapter we are going to introduce some basic terminology of RL that will help us explain the methodologies being used most in RL currently.

3.1.1 MDPs

The simple figure below describes the interaction between an agent and the environment that he's in, that's modelled like a Markov Decision Process.

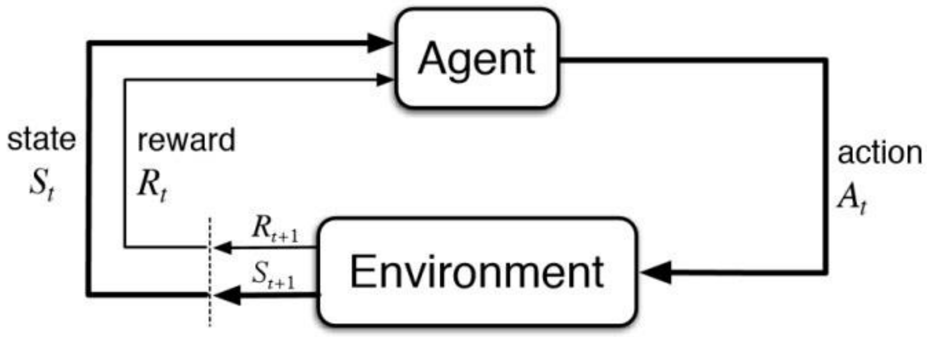


Figure 3.1: A typical RL scenario[4]

A MDP consists of the following tuple (S,A,R_a, P_a ,γ) where:

- S is the set of states in the environment
- A is the set of actions in the environment
- R_a is the reward received by performing action a in a given state
- P_a is the probability of getting to a state, given a previous state and an action
- γ is the discount factor for future rewards, that makes earlier rewards more appealing

The most basic property of MDPs is the "Markov" property, which is about the conclusion that any given process X_{t+1} is dependent upon the previous timestep X_t that coincides with every timestep before that. So in general, we only care about the state before and not of all the previous states

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_t, S_{t-1}, \dots, S_1]$$

MDPs in RL can be without knowledge (model-free) or with full knowledge of the full transition dynamics and rewards (model-based). Model-free methods are solved via trying to figure out the environment's principles, while model-based can be solved by Dynamic programming. So, we categorize RL methods into those that rely on the model and those who do not.

What we are trying to calculate in RL environments is the policy π(s) , which might be deterministic for each state $\pi(s) = a$, or might be stochastic $\pi(a|s) = P_\pi(A = a|S = s)$ and the value function $V_\pi(s)$, which is the expected amount of rewards being in a state and acting according to a policy. But how do we evaluate a value function?

3.1.2 Bellman Equations

A value function under a policy π is the expected return when starting in a state $s \in S$ and following that policy. So we can express it as :

$$V^\pi(s) = E_\pi[R_t|s_t = s] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|\, s_t = s\right] \qquad \textbf{Eq 3.1}$$

and it represents the expected value of discounted rewards that we can get from this state. The discount factor γ used here , which is bounded between 0 and 1 is a metric for how much we care about immediate or future rewards. If γ approximates 0 then we prioritize immediate rewards, while if γ approximates 1 then we place great value on future rewards and long term thinking. We generally prefer immediate rewards over those further in time, from both a natural standpoint because later rewards tend to be more uncertain and a mathematical standpoint because γ helps convergence instead of having to keep track of a large number of future steps.

Another term that is used extensively in RL is the state-action value function Q-value

$$Q^{\pi}(s,a) = E_{\pi}[R_t | s_t = s, a_t = a] = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | \, s_t = s \,, a_t = a\right] \qquad \textbf{Eq 3.2}$$

, which is the expected return after being in state s , performing action a and then following policy π. Furthermore, the Q-function and V-function are related to each other by $V_{\pi}(s) = \sum_a \pi(a|s) Q^{\pi}(s,a)$, meaning that the V-value of state is equal to the sum of every state-action value regarding that state multiplied by the probability of picking that action in that state. We can take the equation a step further, showcasing its recursive element by

$$V^{\pi}(s) = \sum_a \pi(a|s) \sum_{s'} P^a_{ss'} \left(r(s,a) + \gamma V^{\pi}(s')\right) \qquad \textbf{Eq 3.3}$$

This important equation is called the Bellman equation for V_{π}.

Finally, action value $Q^{\pi}(s,a) = \sum_{s'} P^a_{ss'} \left(r(s,a) + \gamma V^{\pi}(s')\right)$, with $P^a_{ss'}$ being the probability of going from state s to state s' by following action a, $r(s,a)$ being the reward from following that action and $\gamma V_{\pi}(s')$ being the recursive discounted term. We can also describe recursiveness in Q-function by the Bellman equation for Q_{π}

$$Q^{\pi}(s,a) = \sum_{s'} P^a_{ss'} \left(r(s,a) + \gamma \sum_{a'} \pi(a'|s') Q^{\pi}(s',a')\right) \qquad \textbf{Eq 3.4}$$

3.1.3 Bellman Optimality Equations

In RL our goal is to find the optimal policy, which is achieved by picking a policy that has the highest expected return than every other policy. So it's obvious that $\pi > \pi'$ iff $V^{\pi}(s) > V^{\pi}(s')$ for all $s \in S$. We define the optimal policy as π^* (we may have more than 1 optimal policy) and its optimal value and state-value function as :

$$V^*(s) = \max_{\pi} V^{\pi}(s) \qquad\qquad Q^*(s) = \max_{\pi} Q^{\pi}(s)$$

We can reuse our previous Bellman equations in order to derive the optimal ones. So the Bellman optimality equation for V^* , using Eq 3.3, is:

$$V^*(s) = \max_{a} \sum_{s'} P^a_{ss'} \left(r(s,a) + \gamma V^*(s')\right) \qquad \textbf{Eq 3.5}$$

and for Q*, using Eq 3.4 :

$$Q^*(s,a) = \sum_{s'} P^a_{ss'} \left(r(s,a) + \gamma \max_{a'} Q^*(s',a')\right) \qquad \textbf{Eq 3.6}$$

As we can observe, summation of the policy from Eqs 3.3 and 3.4 is being replaced with the max term, which picks the action that maximises the value and action value functions respectively.

If we have complete information of the environment, then we are able to solve the problem with Dynamic Programming but in most cases this is not doable, as we are usually not provided with the dynamics of it.

3.1.4 On-policy vs Off-policy

One of the most important concepts in RL is the distinction between on-policy , off-policy and offline algorithms. In general, when our agent acts, according to a behaviour policy we established, he receives a reward. Based on the action picked by the behaviour policy, the reward he received and the next state, we update its Q- values ($\mathrm{Q(s,a)}$). That is what we call an on-policy algorithm, because we use the same action as the one picked by the behaviour policy , in order to update our policy. However, off-policy algorithms use actions to update the agent's policy , that might be independent of those that were picked. For example, we may follow an epsilon-greedy algorithm to select the action and we may update our policy with an action picked by a different algorithm, i.e the action that maximizes Q(s,a). Lastly, offline RL algorithms is another subdivision of algorithms that have no interaction with the environment , unlike on-policy and off-policy, and they are provided with a static dataset , which they must utilize for the purpose of finding a policy. These algorithms resemble the supervised learning methods we saw, with the static dataset being our policy training set. We will take a closer look into offline RL in Chapter 4.

3.1.5 Prediction and Control

A prediction task in RL is the process of estimating the value function when having a fixed policy $\pi(a|s)$. That means that we are interested in predicting if the policy behaves adequately by calculating these state values. On the other hand, a control task is the task of approximating the optimal policy, in order to get the maximum expected reward from each state. As we will later see in GPI (Generalized Policy Iteration), both the policy and the value function are approximated concurrently , with the value function approximating the policy and the policy improving by usually maximizing the value function.

3.1.6 Behavioral Cloning (BC)

Behavioral cloning is a way of learning using the training data provided by a demonstrator[9] and then training a classifier or regressor to match the original behaviour policy. So, there is a dependence between the demonstrator and the trained policy, in the sense that if the data provided is not optimal, then the extracted policy will most likely be suboptimal also. It is mostly used in the form of images where a CNN extracts the information and uses it to classify its actions that are available. It has had its fair share of success in the fields of autonomous driving.

3.2 Value-based methods

As we described earlier, in a RL environment we are trying to optimize our agent's policy or value function. We can actually focus on modelling the policy with the policy-based methods, modelling the value function with the value-based methods or modelling both with actor-critic methods. So, the value-based methods do not explicitly instruct the agent on what action to pick but rather update the value function on the states, so he is aware of each state's utility. Let's analyze some of these methods.

3.2.1 Dynamic Programming

Dynamic Programming assumes that the model is known[3]and uses Bellman equations in order to derive the optimal value function and subsequently the policy. It comprises of 3 parts, that form the algorithm GPI. First, we have the policy evaluation, which uses Eq 3.3 so it can compute the value function for a given policy π. The next part is the policy improvement, where we act greedily as a means to improve our value function. That makes sense because if we set $\pi^{'}(s) = argmax_a q_\pi(s,a)$ then our action value function $q_\pi\left(s, \pi(s^{'})\right)$ should be greater or equal to the action value function using the given policy. In other words,

$$q_\pi\big(s,\pi(s')\big) = \max_{\mathrm{a}} q_\pi(s,a) \geq q_\pi\big(s,\pi(s)\big) = v_\pi(s)$$

But the term $q_\pi\big(s,\pi(s')\big)$ can be decomposed recursively into a value function, as it constitutes the expectation of following policy π' in state s and then following policy π afterwards . So,

$$\begin{aligned} q_\pi\big(s,\pi(s')\big) = {} & \mathrm{E}_{\pi'}[\mathrm{R}_{t+1} + \gamma V_\pi(S_{t+1}) |\, S_t = s] \\ \leq {} & \mathrm{E}_{\pi'}\big[\mathrm{R}_{t+1} + \gamma q_\pi\big(S_{t+1}, \pi'(S_{t+1})\big) |\, S_t = s\big] \\ \leq {} & \mathrm{E}_{\pi'}\big[\mathrm{R}_{t+1} + \gamma \mathrm{R}_{t+2} + \gamma^2 q_\pi\big(S_{t+2}, \pi'(S_{t+2})\big) |\, S_t = s\big] \\ \leq {} & \mathrm{E}_{\pi'}[\mathrm{R}_{t+1} + \gamma \mathrm{R}_{t+2} + \gamma^3 R_{t+3} + \cdots |\, S_t = s] = V_{\pi'}(s) \end{aligned}$$

Which means that $V_\pi(s) \leq V_\pi'(s)$. Improvement in the policy can happen until the state-action value of the greedy policy is equal to the state-action value of our policy:

$$q_\pi\big(s,\pi(s')\big) = \max_{\mathrm{a}} q_\pi(s,a) = q_\pi\big(s,\pi(s)\big) = v_\pi(s)$$

,then we have the optimal policy π' . The last part is the policy iteration , where we combine the first 2 parts , as we first evaluate the policy by computing the value function and then we improve it by applying the greedy policy.

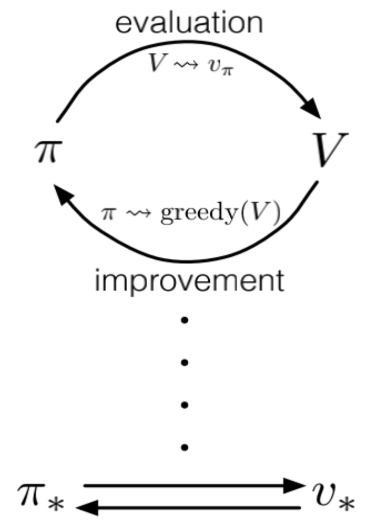

evaluation
$V \rightsquigarrow v_\pi$
π
V
$\pi \rightsquigarrow \text{greedy}(V)$
improvement
π_*
v_*

Figure 3.2 : Policy iteration algorithm in DP [8].

3.2.2 Monte-Carlo methods

Monte-Carlo methods are methods that take advantage of the full episode of states, actions and rewards until it is finished. They are model-free so the only way to collect information about the environment is by interacting with it and they don't require any knowledge of the MDP but can only be applied to episodic MDPs , i.e MDPs that finish. So we know that $V_\pi(s) = E_\pi[G_t|S_t = s]$, where $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$ is the total sum of the discounted rewards. The basic idea of MC methods is that the average returns after visiting a state should converge to the expected value of the state.

MC prediction methods are split into 2 categories: first-visit MC and every-visit MC. The first category refers to the estimation of $V(s)$ by averaging the returns after the first visit of state s, while the second category calculates $V(s)$ by averaging the returns after every visit of state s. Every time a state s is visited we update the number of visits to that state , $N(s) = N(s) + 1$ and we update the returns after state s , $R(s) = R(s) + G_t$. Then we calculate the value of the state by the mean return $V(s) = \frac{R(s)}{N(s)}$

A very useful way in terms of memory that we can use MC is using the incremental implementation. For example, if we want to average some values then:

$$\mu_n = \frac{1}{n}\sum_{i=1}^{n} y_i = \frac{1}{n}(\sum_{i=1}^{n-1} y_i + y_n)$$
$$= \frac{1}{n}(\,(n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} y_i + y_n)$$
$$= \frac{1}{n}(\,(n-1)\mu_{n-1} + y_n) = \mu_{n-1} - \frac{1}{n}\mu_{n-1} + \frac{1}{n}y_n = \mu_{n-1} + \frac{1}{n}(y_n - \mu_{n-1})$$

Now using the value functions instead we get :

$$V(S_t) \leftarrow V(S_t) + a\big(G_t - V(S_t)\big) \qquad \textbf{Eq. 3. 7}$$

,where a is a constant with a value between $(0,1]$. So, in order to update our value function , we only need our previous V-function for the state and the cumulated reward after timestep t.

As with DP we can use policy iteration for MC control , with the exception of using the state action values $\mathrm{Q(s,a)}$ instead of the state values as the description of the model is unknown. So again our policy iteration algorithms is comprised of the 3 parts, policy evaluation, policy improvement and policy iteration. In the policy evaluation step, after we have initialized a policy π, we repeat a designated number of episodes in which we use our policy π to perform actions and calculate the total reward $R(s,a)$ for every episode. After we've finished the episodes, we compute the Q-value of each state-action pair with the average of $R(s,a)$ starting from that pair. Then, on the policy improvement step, we improve the policy by picking the greedy actions $\pi'(s) = argmax_a Q(s,a)$. The policy iteration part is where we alternate between evaluation and improvement until π converges to π*.

A problem unravelling with that approach is that many states might not get visited because of the greediness of the new policy π'. A solution to this would be exploring starts , i.e to grant each state action pair a non-zero probability of being the starting pair. That way, more exploration is ensured. Another way to achieve this without having to start with random pairs is to just use an epsilon-soft policy:

$$\pi(s) = \begin{cases} 1-\varepsilon+\dfrac{\varepsilon}{A(S_t)}, & if\ a = A^* \\ \dfrac{\varepsilon}{A(S_t)}, & if\ a \neq A^* \end{cases}$$

, where $A(S_t)$ is the total number of actions that can be performed from state S_t . So , this softmax policy allows each action to have a non-zero probability of being chosen.

In general, on a model-free environment, MC can work well if the state-action space is not immense, as that would require too much computation power for every state-action pair to be visited.

3.2.3 Temporal-Difference Learning

TD learning resembles MC methods at being model-free and learning from episodes, but the advantage over MC is that with TD we can learn from incomplete episodes, so we do not need a full trajectory in order to update our Q-function. The only thing that differs from the MC incremental implemention in Eq. 3.7 is the substitution of G_t:

$$V(S_t) \leftarrow V(S_t) + a\big(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\big) \qquad \mathbf{Eq. 3.8}$$

,which is called bootstrapping and refers to updating the target with the use of existing estimates as opposed to complete trajectories. We know that $G_t = R_{t+1} + \gamma V(S_{t+1})$, because

$$V_\pi(s) \approx E_\pi[G_t | S_t = s] = E_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] = E_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s]$$

Eq. 3.8 refers to the TD(0) method, which looks one step into the future. But what if we want to make more steps into the future? We could, for instance, take an n-step TD:

$$R_t^n = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n})$$

But another justifiable question is could we combine multiple n-step returns?

For example, we may want to backup half of a 3-step return and half of a 4-step return. That’s where TD(λ) comes into play.

TD(λ) is a generalization of TD(0) and MC methods that looks an arbitrary amount of steps into the future. It works by averaging each of the n-step updates, weighted by $\lambda^{n-1}, \lambda \in [0,1]$ and is normalized with $(1-\lambda)$ so the sum cannot exceed 1. It is described by the following equation:

$$G_t^{\lambda} \approx (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} + \lambda^{T-t-1} G_t \qquad \mathbf{Eq. 3.9}$$

So, the weight of each n-step return gets smaller as time progresses. The second term in Eq 3.9 refers to the all of the n-returns that happen after we reach a terminal state. We can observe here that for the maxima points of λ , that is λ = 0 and λ = 1 , we get the one-step TD method and the MC method respectively. However, there is a problem with the previous approach, called forward view , and that is the need to look forward in order to update each state, which causes the updates to become off-line and take place at the end of the episode. This issue is tackled utilizing the backward view.

The backward view is associated with eligibility traces, a variable that keeps up with the state's visitation. The formula for the eligibility trace is:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s), & if\ s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1, & if\ s = s_t \end{cases} \qquad \mathbf{Eq. 3.10}$$

This is telling us that whenever we visit a state the eligibility trace of that state increases. But, every step that we do not visit a state , that state decreases by the trace-decay parameter λ. So, the frequency and the recency of state visitation comes into play with Eq 3.10. The way we use the eligibility trace in updating the value functions is by placing it into the Eq. 3.8:

$$V(S_t) \leftarrow V(S_t) + a\big(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\big)e_t(s)$$

This is the backward view, because it's like we are observing the state visitation backwards and depending on the value of the eligibility trace , we perform a bigger or smaller update step.

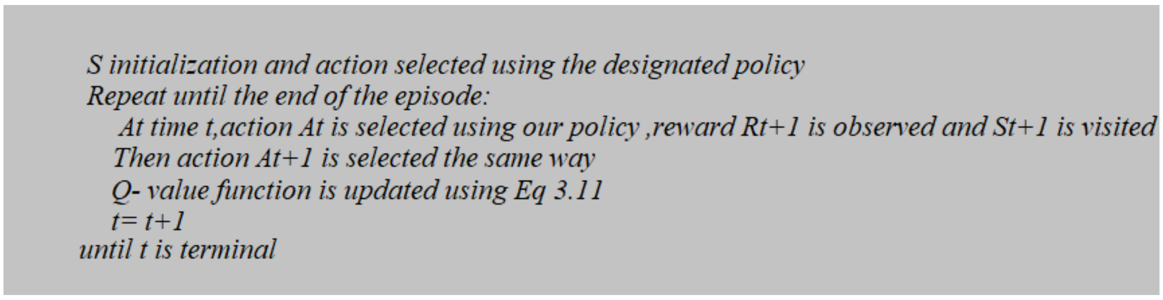
This was a review of the model-free TD prediction method, which evaluates the state functions. But as always, we also care about the control as a way to improve both the value and the policy functions.

3.2.4 SARSA

Model-free control is about optimizing the value and the policy function for an unknown MDP. Value functions cannot improve a policy in a model-free environment, as the probability transitions and the maximum rewards are not known for each state. That leads us to using action-state values in order to improve the policy. Thus, the control equivalent in TD prediction is a variation of the Eq 3.8:

$$Q(S,A) \leftarrow Q(S,A) + a\big(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S,A)\big) \qquad \boldsymbol{Eq\ 3.11}$$

SARSA is an on-policy algorithm, because it uses the action selected to update the Q-values , as the Figure 3.3 shows.



```
S initialization and action selected using the designated policy
Repeat until the end of the episode:
    At time t,action At is selected using our policy ,reward Rt+1 is observed and St+1 is visited
    Then action At+1 is selected the same way
    Q- value function is updated using Eq 3.11
    t= t+1
until t is terminal
```

Figure 3.3 : Sarsa Pseudocode

A policy that's often employed for the purpose of selecting actions is the epsilon soft algorithm that was described in section 3.2.2. SARSA(λ) is defined as well, by applying Eq. 3.9 into our SARSA update:

$$Q(S,A) \leftarrow Q(S,A) + a\left(q_t^{\lambda} - Q(S,A)\right) \qquad \boldsymbol{Eq\ 3.12}$$

3.2.5 Q-learning

Q-learning is an algorithm very similar to SARSA , however it possesses a significant difference, the off-policy property. That means that whatever action we choose, does not matter into the update rule. For example, we could perform an ε-soft policy for the action selection and a greedy algorithm for the update rule. The update rule of Q-learning exhibits that:

$$Q(S,A) \leftarrow Q(S,A) + a\left(R_{t+1} + \gamma \max_{\alpha} Q(S_{t+1},\alpha) - Q(S,A)\right) \qquad \boldsymbol{Eq\ 3.13}$$

The difference is obvious and it is the reason of the separation in notion between on policy and off policy algorithms. Still, the most sensational breakthrough in RL algorithms came into being with the fusion of deep learning and RL (in this case specifically Q-learning).

3.2.6 Deep Q network

An issue that arises with the methods of updating the Q values, is that in environments where the state and action spaces are immense, the process of updating the whole table of values is practically infeasible. A solution to this are function approximators and as we've seen in the previous chapter, NNs can help in that.

The algorithm is basically Q-learning performed in a deep learning setting, with added features that result in a method that has had much success in the state action spaces we discussed above. One of these features is the memory replay dataset [38]. DQN utilizes the replay memory dataset in order to remove correlations. Sequential data like the data provided in RL environments are very likely to be correlated, as the next state is affected by the action taken and thus handling experiences sequentially results in instabilities regarding the function approximator, the neural network . In general, neural networks work better with data that are i.i.d, i.e data that are independent and identically distributed without possessing sequential properties. But there seems to be a hindrance regarding the update of the Q-functions. We saw that in Q-learning the update was directed by the term in parentheses in Eq. 3.13, which is the target value (the value we get using the recursive property of Q-functions) minus our estimate of the Q-function in the state we are into and the action that is chosen. But if we are to perform updates the same way in DQN, then we would be updating every Q value in each iteration update, because the weights of the NN would constantly change. The solution to this is referred at [39], and is the parameterization of the target network with a different parameter than the one used in updating the Q function. This leads to certain benefits from a stability viewpoint, as now the target network resembles a supervised learning target that remains mostly fixed throughout the update phase. The parameter of the target network is, let's say, θ^- with θ^- being different than θ. The idea that makes the parameters of the two networks stay relevant is that the parameters of the prediction network are copied into the target network every predefined number of steps, for example C. So, while the prediction network constantly changes values, the target network stays fixed for C steps and then basically takes on the prediction network values. Thus, the loss function is defined like:

$$L = (r + \gamma \max_{\alpha'} Q(s', a'; \theta') - Q(s, a; \theta))^2$$

And the parameter update rule becomes:

$$\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta E_{s' \sim P(s'|s,a)} [(r + \gamma \max_{\alpha'} Q_k(s', a'; \theta') - Q_k(s, a; \theta)]$$

3.3 Policy-based methods

3.3.1 Introduction

These kind of methods are aiming towards learning the policy directly without learning the value function. They are generally more useful in cases where the number of states and actions are vast and value-based methods cannot compete, since they require a full scan of the space. The "wellness" of a policy is measured based on the context. For instance, in episodic environments a metric of a policy would be $J(\theta) = V^{\pi_\theta}(s_0)$. But in continuing environments the metric is the average value:

$$J(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s) = \sum_s d^{\pi_\theta}(s) \sum_\alpha \pi_\theta(\alpha|s) Q^\pi(s, a) \qquad \boldsymbol{Eq\ 3.14}$$

, where $d^{\pi_\theta}(s)$ is the stationary distribution of Markov chain for the policy π_θ . In other words, this distribution is the probability that we end up with state s_t when starting from a state s_0 and following policy π_θ for t number of steps:

$$d^{\pi}(s) = \lim_{t\to\infty} P(s_t = s|s_0, \pi_\theta)$$

The basic goal here is to calculate the gradient of our reward function $J(\theta)$ in respect to the parameter θ and move towards the direction that maximizes the function. This approach in policy based methods is effective in high-dimensional spaces and has better convergence properties but a common problem is that with the gradient ascent that we perform on our reward function, we converge to a local and not a global maximum.

3.3.2 Policy Gradient Theorem

The calculation of the gradient $\nabla_\theta J(\theta)$ is tricky because it involves differentiating the stationary distribution and the value function. Below the process of computing that gradient is displayed, following the policy gradient theorem[2]:

First we perform the gradient of the value function:

$$\begin{aligned}
\nabla V_\pi(s) &= \nabla\left[\sum_{\text{a}} \pi(a|s) q_\pi(s,a)\right] \\
&= \sum_{\text{a}} [\nabla\pi(a|s) q_\pi(s,a) + \pi(a|s)\nabla q_\pi(s,a)\,] \\
&= \sum_{\text{a}} \left[\nabla\pi(a|s) q_\pi(s,a) + \pi(a|s)\nabla \sum_{\text{s}',\text{r}} \text{p}(\text{s}',\text{r}|s,a)(r + V_\pi{}'(s)) \quad\right] \\
&= \sum_{\text{a}} \left[\nabla\pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{\text{s}'} \text{p}(\text{s}'|s,a)\nabla V_\pi{}'(s) \quad\right] (1) \quad \boldsymbol{Eq\ 3.15}
\end{aligned}$$

In step (1) we remove the rewards because $\text{p}(\text{s}'|s,a) = \sum_r p(s',r|s,a)$ *which means that the probability of getting into a next state is the sum of probabilities of all the next state transitions along with their rewards. So if there are 2 transitions to the same state, we sum both of their probabilities based on the different rewards.*

As we see, Eq. 3.15 has a recursive form so the gradient of the value function can be unrolled into future states. Now we need to define the probability of going from state s to state x in k steps using policy π, which is itself recursive because for k = 1 :

$$\rho^{\pi}(s \to s', k = 1, \pi) = \sum_{a} \pi_\theta(\alpha|s) P(s'|s,a)$$

That equation takes into account every action that can lead to state s'. So the general recursive equation of ρ^{π} is:

$$\rho^{\pi}(s \to x, k+1, \pi) = \sum_{s'} \rho^{\pi}(s \to s', k)\rho^{\pi}(s' \to x, k=1) \qquad \boldsymbol{Eq\ 3.16}$$

The general equation here shows its recursiveness in its first term which will unroll until we reach state s'. So using that equation in unrolling the value function of the next states in Eq. 3.15, we derive the following equation:

$$\begin{aligned} \nabla_{\theta} J(\theta) &= \nabla V_{\pi}(s_0) \\ &= \sum_{s} \sum_{k=0}^{\infty} \rho^{\pi}(s_0 \to s, k, \pi) \sum_{\mathrm{a}} \pi(a|s) q_{\pi}(s,a) \\ &= \sum_{s} \eta(s) \sum_{\mathrm{a}} \pi(a|s) q_{\pi}(s,a) \\ &= (\sum_{s} \eta(s)) \frac{\eta(s)}{\sum_s \eta(s)} \sum_{\mathrm{a}} \pi(a|s) q_{\pi}(s,a)\ , \end{aligned}$$

which is analogous to $\frac{\eta(s)}{\sum_s \eta(s)} \sum_{\mathrm{a}} \pi(a|s) q_{\pi}(s,a)$ since $\sum_s \eta(s)$ is a constant. Therefore:

$$\begin{aligned} \nabla_{\theta} J(\theta) &\propto \sum_{s} d^{\pi}(s) \sum_{\mathrm{a}} \pi(a|s) q_{\pi}(s,a) \\ &= \sum_{s} d^{\pi}(s) \sum_{\mathrm{a}} \pi_{\theta}(a|s) q_{\pi}(s,a) \frac{\nabla_{\theta} \pi_{\theta}(a|s)}{\pi_{\theta}(a|s)} \\ &= E_{\pi}[Q^{\pi}(s,a) \nabla_{\theta} \ln \pi_{\theta}(a|s)] \qquad \boldsymbol{Eq\ 3.17} \end{aligned}$$

In the above equation $\frac{\eta(s)}{\sum_s \eta(s)} = d^{\pi}(s)$ because it is a stationary distribution. Eq 3.17 is the basis for a lot of policy based algorithms. In the next subsection we will introduce a policy based algorithm.

3.3.3 REINFORCE (Monte Carlo policy gradient)

In this algorithm the approach is very similar to our value-based MC[2], since we are using a full trajectory, in order to update our θ parameter of our reward function. In other words:

$$\nabla_\theta J(\theta) = E_\pi[Q^\pi(s,a)\nabla_\theta \ln \pi_\theta(a|s)] = E_\pi[G_t \nabla_\theta \ln \pi_\theta(a|s)] \quad ,$$

as $E_\pi[G_t|s_t, a_t] = \; Q^\pi(s_t, a_t)$

So the whole idea is to sample a full trajectory using an arbitrary policy θ as in $\pi_\theta: s_1, a_1, r_2, s_2, a_2, \dots s_T$ and then for every timestep $t = 1, ., ., T$ the return G_t is estimated and the parameter θ is updated by:

$$\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta \ln \pi_\theta(a|s) \qquad \boldsymbol{Eq\ 3.18}$$

A problem with the approach of a full trajectory is the high variance that is apparent. A common approach to this issue is to introduce an advantage function[1]. A simple and common used example of advantage function is subtracting the state value from the action value as in $A(s,a) = Q(s,a) - \, V(s)$ and using it instead in place of G_t. Here the state value is called a baseline function, that generally serves as a basis for the actions that are being picked, with the advantage being large in actions that deviate from the average value of the state function.

3.4 Actor-Critic methods

One of the most used methodologies in RL is the actor critic and for a good reason. It basically combines the policy based aspect with the value function approach. The way this is implemented is by the usage of 2 models, the critic model and the actor model. The critic model keeps up with updating the value function parameters while the actor model is busy with updating the policy in the way that is proposed by the critic, as can be seen in figure 3.4. We usually model this with two separate parameterized networks, one that keeps up with updating the policy, which is the actor and the other with updating the value function which is the critic. Actor critic methods are composed of the policy update, following the action that was sampled from our policy π_θ, then the TD error is computed for the action value function at the given state-action pair and that particular error is used to update the parameters of our critic network. So the critic works towards helping the actor on improving its policy by updating the Q function parameters which are used in the update of the policy π_θ.

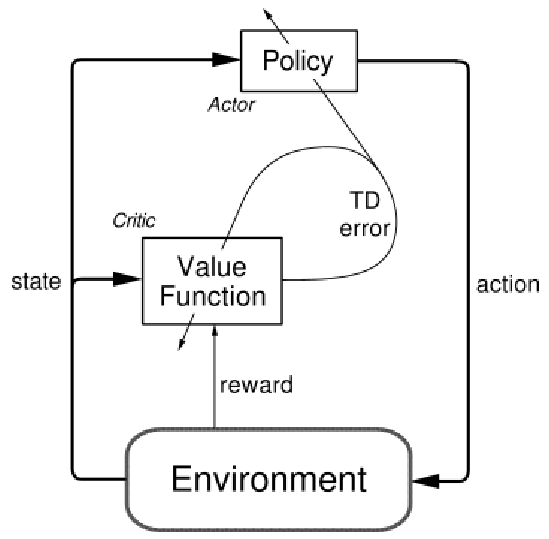


Figure 3.4 : Actor-critic network presentation[4].

3.4.1 Standard AC

The way the standard actor critic works in practice is by sampling an action a, followed by sampling the given reward r and the next state s' and then sampling again a' given our current state. What this means is that our introductory steps are exactly the same as SARSA and Q learning but the main difference is apparent in the next step. After sampling states and actions, the derivative of the actor-critic cost function J is defined by following Eq. 3.17 and replacing the Q-function with a parameter w that's being monitored by a neural network. The resulting equation is

$$\nabla_\theta J(\theta) = E_\pi[Q_\omega{}^\pi(s,a)\nabla_\theta \ln \pi_\theta(a|s)]$$

and then the update on the policy parameter θ is similar to Eq 3.18, but involves the parameterized Q function instead of the trajectory G_t.

$$\theta \leftarrow \theta + \alpha Q_\omega{}^\pi(s,a)\nabla_\theta \ln \pi_\theta(a|s)$$

After we updated the policy parameter, we still have to update the Q-function parameter w aswell as compute the TD error regarding the actions that were sampled. So, the correction error is computed as normal

$$\delta_t = r_t + \gamma Q_\omega^\pi(s',a') - Q_\omega^\pi(s,a)$$

Afterwards, the only thing that's left is to update the Q-function parameters with the computed TD error in mind

$$\omega = \omega + \alpha\delta_t \nabla_\omega Q_\omega^\pi(s,a)$$

That is the standard actor-critic algorithm that trains 2 parameters, namely ω and θ in order to unravel the optimal policy.

3.4.2 A3C and A2C

But there are also variations of this algorithm that aim at reducing the variance by replacing the Q function with the so called advantage term that we introduced earlier in 3.3.3. Specifically, Eq. 17 now turns into:

$$\nabla_\theta J(\theta) = E_\pi[A(s,a)\nabla_\theta \ln \pi_\theta(a|s)]$$

We've seen the advantage formula before but it poses a problem. The difference between the Q function with a designated state-action pair and the value function of a state, requires the approximation of them both by a neural network. Since that would be extremely cost inefficient we can use the Bellman equations established in the introductory chapter 3.1, as a means to end up with one parameterized function. Indeed:

$$A(s_t\ , a_t) = Q_\omega(s_t, a_t) - V_\kappa(s_t) = r_{t+1} + \gamma V_\kappa(s_{t+1}) - V_\kappa(s_t)$$

The algorithm that employs this technique is called the Asynchronous Advantage Actor Critic(A3C) and is an algorithm that employs several asynchronous actor-learners to train in parallel, by utilizing CPU threads keeping the training on the same machine[29]. The advantage of training multiple learners is that the diversity of the chosen actions is big and the updates occurring in the parameters are less likely to be correlated over time, especially if each actor possesses different exploration policies which results in exploring even more. So, in contrast to the DQN employing a replay memory dataset, the multiple actors here are in charge of the stabilizing element of the function approximator that is otherwise induced by the replay dataset. The main problem though, is that asynchrony means that some of the actors that are not updated as frequently, will have to use older versions of the network parameter. That's why a variation of the A3C was introduced, the

A2C algorithm that performs synchronous updates (removing the asynchronous aspect of A3C) , meaning that it waits for all the actors to finish the segment of experience specified before executing the update, which is basically the average over all the actors. But In general, A2C seems to enjoy more success than its asynchronous counterpart as specified in [30] , mainly because of the more effective usage of GPUs by the A2C algorithm.

3.4.3 SAC

But one of the most used variations of the standard Actor Critic methods is the SAC (Soft Actor Critic) that's being used in alot of experiments involving both online and offline RL, being compared to other algorithms in terms of expected returns. The interesting part about SAC is that in addition to the standard expected return, an entropy regularization term is appended to the formula. Entropy represents the randomness of a variable in a given distribution, so we can intuitively guess that the more entropy we have, the more exploration will be achieved in our environment. That brings a change to the main RL problem, which is maximizing the policy in order to get the maximum reward:

$$\pi^* = argmax_\pi J(\pi), \qquad where\ J(\pi) = E_{\tau\sim\pi}[R(\tau)]$$

Now instead of that we get the following equation:

$$\pi^* = argmax_\pi E_{\tau\sim\pi}[\sum_{t=0}^{\infty} \gamma^t (R(s_t, a_t, s_{t+1}) + \alpha H(\pi(\cdot\,|s_t)))] \quad \boldsymbol{Eq\ 3.19}$$

With H being the entropy term $H(X) = -\sum_{i=1}^{n} P(x_i)\log\, P(x_i)$ and α being the coefficient that regulates the importance of the entropy term, better known as temperature parameter. Likewise the value and state value functions are formulated:

$$V^\pi(s) = E_{\tau\sim\pi}[\sum_{t=0}^{\infty} \gamma^t \left(R(s_t, a_t, s_{t+1}) + \alpha H(\pi(\cdot\,|s_t))\right) |s_0 = s]$$

$$Q^\pi(s,a) = E_{\tau\sim\pi}[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) + \sum_{t=1}^{\infty} \alpha H(\pi(\cdot\,|s_t))\,|s_0 = s, a_0 = a]$$

With the only difference in the action value function being that we evaluate the entropy term starting from timestep $t = 1$ [33]. The layout of the algorithm is the policy parameterized by , π_θ , the soft Q-value that's parameterized by ω, Q_ω and the soft state value function that's parameterized by ψ, V_ψ. Although the policy and Q-value function are sufficient enough for us to deduce the state-value function V, the algorithm still uses a NN to approximate it for stability reasons during the training. In [32] the soft value function becomes:

$$V(s_t) = E_{a_t \sim \pi}[Q(s_t, a_t) - \log \pi(a_t|s_t)] \quad \boldsymbol{Eq\ 3.20}$$

, where the value function is contained inside the Q-function according to the Bellman equations:

$$Q(s_t, a_t) = r(s_t, a_t) + \gamma E_{s_{t+1} \sim p}[V(s_{t+1})] \qquad \boldsymbol{Eq\ 3.21}$$

Now that our value functions are defined, we can delve deeper into the algorithm described by the original paper [35]. The cost function of the value function with respect to parameter ψ is:

$$J_V(\psi) = E_{s_t \sim D}[\frac{1}{2}\left(V_\psi(s_t) - E[Q_\omega(s_t, a_t) - log\pi_\theta(a_t|s_t)]\right)^2]$$

$$with\ the\ gradient\ being\ \nabla_\psi J_V(\psi) = \nabla_\psi V_\psi(s_t)(V_\psi(s_t) - Q_\omega(s_t, a_t) - log\pi_\theta(a_t|s_t))$$

The D in the cost function of the value function is the replay buffer that is used to store states and actions. Similarly we can define the cost function of the action value function:

$$J_Q(\omega) = E_{(s_t, a_t) \sim D}[\frac{1}{2}\left(Q_\omega(s_t, a_t) - r(s_t, a_t) + \gamma E_{s_{t+1} \sim p}[V_{\psi'}(s_{t+1})\right)^2$$

$$with\ the\ gradient\ being \nabla_\omega J_Q(\omega) = \nabla_\omega Q_\omega(s_t, a_t)(Q_\omega(s_t, a_t) - r(s_t, a_t) + \gamma V_{\psi'}(s_{t+1}))$$

In this case of the action value cost function we can see that a new variable is introduced, ψ', which is the parameter of a target value function $V_{\psi'}$. That's called the exponentially moving average of the value network weights and it's used to improve stability because of the reason that the weights of our network are updated according to it. A similar logic is followed in the DQN case that was covered in earlier chapters, with the parameter of the target Q network. Lastly, the policy is updated according to the KL divergence, which is a metric of the difference between two distributions. Here we would like to minimize the KL divergence between our policy and the exponent of the Q-function divided by a normalization function Z_θ. So the cost function of our policy is :

$$J_\pi(\theta) = E_{s_t \sim D}[D_{KL}(\pi_\theta(\cdot\,|s_t)|| \frac{\exp\left(Q_\omega(s_t,\cdot)\right)}{Z_\omega(s_t)}]$$

,which is then minimized according to the gradient. So, after all these steps that every cost function was defined, the process is really simple. At first, the parameters $\theta, \omega\ and\ \psi, \psi'$ are initialized along with their respective learning rates $\lambda_\pi, \lambda_Q, \lambda_V$. After that, for each step in the environment, we sample an action a from our policy π_θ given a particular state (starting from s_0), receive the reward from executing that action and sample the next state s_{t+1} using the transition probability $p(s_{t+1}|s_t, a_t)$. This tuple is then appended to our replay buffer as a stored experience tuple:

$$D \leftarrow D \cup \{(s_t, a_t, r(s_t, a_t), s_{t+1}\}$$

Then after we've taken some steps and want to update the parameters we've previously defined, we just update them by subtracting them with their learning rate multiplied by their respective cost function. Only the update of the ψ' parameter is different because it is special case parameter and is defined as:

$$\psi' = \tau\psi + (1-\tau)\psi'\ ,$$

$$where\ \tau\ is\ the\ exponential\ moving\ average\ factor$$

SAC is in general a fairly complex algorithm that with the addition of the entropy factor manages to achieve state of the art results in experiments conducted.

CHAPTER 4

OFFLINE REINFORCEMENT LEARNING

4.1 Introduction

A real problem with RL is the fact that generalization is hard to achieve without many simulations. We really do not fancy having to perform simulations upon simulations in order to improve the policy, especially in real world environments, where the data collection procedure is costly and time-consuming. What paves the way for generalization ,as we know from machine learning, are large static datasets that contain information about the environment we are interested into. But how can we import large datasets into the RL world? That is achieved with offline RL, which is the topic for discussion in this thesis.

We have previously covered the on-policy and off-policy methods. The on-policy methods operate by performing actions dictated by a designated policy and then using these actions to update the policy. Off-policy choose actions based on a policy and then update that policy with actions that stem from another policy, so they are not policy dependent and issue more exploration. They also utilize a buffer that the states, actions and rewards at each iteration are added into. This buffer, which is dynamic since it is always being refreshed with new entries, serves as training examples for our agent in off-policy methods. Meanwhile, offline learning is the process of our agent learning from static datasets, unlike the dynamic buffer, which is constantly being added upon new information. Overall, in offline learning the agent learns a policy from our static dataset without being able to interact with an environment.

However, many hindrances are apparent when working with offline learning and they come up because of the inability of the agent to interact with his environment. One of the difficulties is the static dataset D containing suboptimal data, affecting the training process and as a consequence, the trained policy. But that is something that we cannot control so we will not get into that. Another issue with the offline approach, is the test data being different from the training data. Then, the agent having not seen the test data before, might overestimate the Q values and think it’s performing well but in reality it might choose the wrong actions that will eventually accumulate and form a completely wrong trajectory. That problem is called distribution shift or extrapolation error and it forms a big issue when trying to train an agent offline. The error that's induced by distribution shift is the overestimation of Q values , due to OOD(out of distribution) actions, i.e actions that were not met during the training phase[4][5].In the next chapter, we will discuss ways of dealing with distribution shift.

4.2 Policy constraint methods

A really important concept in reinforcement learning without feedback , is the policy constraint. Policy constraint methods are methods that try to combat the problem of distribution shift, by taking into account both the behaviour policy distribution of the dataset and the policy distribution that's being trained. There are 2 main types of policy constraint methods, support and distribution constraints.

Distribution constraints are generally concerned about our policy distribution being close to the behaviour policy distribution, so the actions picked are more in line with our dataset. Meanwhile, support constraints impel our trained policy π_θ to choose actions where the density is larger. Support of a function is the set of values for which the function is non-zero, so this constraint does not bother with datapoints that are equal to zero. In other words, distribution constraints mainly compel the distribution of the trained policy to be similar to the distribution of the behaviour policy of the dataset, while support constraints constrain the agent into picking actions that the behaviour policy is highly likely to pick given a particular state. As we can observe in Figure 4.1, in the left plot we are presented with out of distribution actions (OOD) in a circle. These actions are not induced by the behaviour policy β that's denoted by the dashed lines, so what they will end up achieving is propagating Q values that are much higher than what they truly are. The distribution constraint on the middle plot constrains the training policy expressed in purple colour to resemble the distribution of the original behaviour policy β , while support constraint on the right constrains our actions to the specific boundary, specified by the yellow lines, of the behaviour distribution. The distributions in yellow colour refer to our training distribution placing above zero probability on actions with non negligible behaviour policy density.

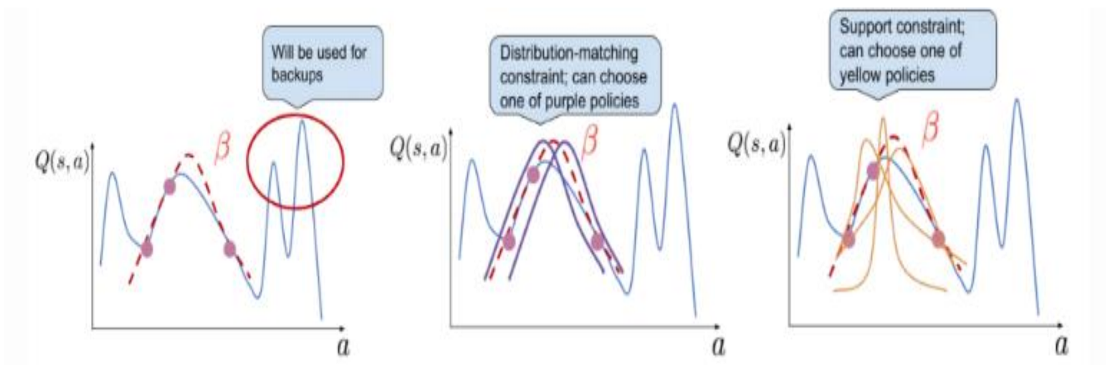

Will be used for backups
Distribution-matching constraint; can choose one of purple policies
Support constraint; can choose one of yellow policies
$Q(s,a)$
β
a

Figure 4.1 : Policy constraint categories[44]

Recent advances have been observed in both categories. Specifically, in the distrib ution constraint area, Batch Constrained deep Q-Learning (BCQ) is an algorithm that attempts to solve the issue of distribution shift. As its name suggests, BCQ constraints the actions picked into being similar with those contained in the batch and afterwards selects the highest-valued action through a Q-network [6]. BCQ resembles Q-Learning but in the update step, instead of choosing the maximum action in a given state , it selects the actions that appear in the batch of data or eliminates actions that are unlikely to be selected by the behaviour policy π_{β}. The algorithm is described below.

4.2.1 BCQ

BCQ trains a variational Autoencoder, from which actions are sampled. This autoencoder tries to approximate the state-conditioned marginal probability $P_B^G(a|s)$. The policy that tries to maximize this probability would in turn minimize the extrapolation error by selecting the most likely actions in our dataset for a particular state. The generative model is used because the probability is difficult to calculate in high-dimensional spaces. Furthermore, a perturbation model is used in order to increase the diversity of the actions being chosen and to prohibit us from having to sample alot of times. The perturbation model is defined as $\xi_{\varphi}(s, a, \Phi)$ and adds to the actions values in the range of $[-\Phi, \Phi]$. So, the process is to sample N actions via the generator, perturb each action using our model and then choose the action with the highest Q-value. The formula of the policy is described below:

$$\pi(s) = argmax_{a_i+\xi_{\varphi(s,a_i,\Phi)}} Q_\theta \left(s, a_i + \xi_\varphi(s, a_i, \Phi)\right), \{a_i \sim G_\omega(s)\}, i = 1, \dots, n \quad \boldsymbol{Eq\ 4.1}$$

As we can observe, the selection of Φ and n gives either behavioral cloning where Φ = 0 and n=1 as we are sampling only 1 action, or Q-learning where $\Phi \rightarrow a_{\{max\}} - a_{\{min\}}$ and → ∞ , meaning that the whole action space is being sampled. Another element of this algorithm is the usage of a modification of Clipped Double Q-learning, which estimates the Q-value by taking the minimum of 2 distinct Q-networks that are being trained. So, to sum up BCQ contains four networks, the generative model $G_\omega(s)$, the perturbation model $\xi_\varphi(s, a)$ and the two Q-networks $Q_{\theta1}, Q_{\theta2}$.

Experiments done using BCQ, show that compared to DQN and DDPG, BCQ approximates the true value way better and does not overestimate its Q-values. Another interesting fact is that in the imperfect demonstration case showcased at the paper, where there is noise in picking actions by the behaviour policy and also noise added in the remaining actions for high exploration, BCQ is able to extract the superior actions and provide a much higher return than the other algorithms. Also, BCQ learns in a small number of steps compared to deep RL methods that require alot of iterations. All in all, BCQ seems to consistently outperform the behaviour policy and the other agents used, except in the imitation learning task where behaviour cloning was in any case expected to come on top.

4.2.2 BEAR

Bootstrapping error accumulation reduction is a support constraint algorithm that is involved into solving the *bootstrapping error*, the accumulated error that's caused by the bootstrapping of actions that lie outside of the training data[4]. The key difference between BEAR and BCQ[6]is that BCQ constrains the distribution of the learned policy to be close to the distribution of the behaviour policy. This is a restriction because for example, if the distribution of the behaviour policy is uniform then the policy will also act uniformly, while with a support constraint, our trained policy will pick the optimal actions with the highest reward in an almost deterministic way.

BEAR is implemented by using K Q-functions and choosing the minimum Q-value for the policy improvement step. Then we maximize the conservative estimate of Q-values by choosing the searching in the policy space, where all the policies share the same support as our behaviour policy. The formula of the new policy improvement step is :

$$\pi_{\varphi} \coloneqq \max_{\pi \in \Delta_{|S|}} E_{s\sim D} E_{a\sim\pi(\cdot|s)}[Q_\theta(s,a)] \quad ,with \;\; E_{s\sim D}[MMD(\beta(.|s), \pi(\cdot\,|s))] \leq \varepsilon \;\; \boldsymbol{Eq\ 4.2}$$

, with ε being an arbitrary value. As described in [4], MMD (maximum mean discrepancy) is implented by using samples from both the behaviour and the trained policies and computing the difference in support between them. MMD is described as:

$$MMD^2(X,Y) = \frac{1}{n^2}\sum_{i,i'} k(x_i, x_{i'}) - \frac{2}{nm}\sum_{i,j} k(x_i, y_j) + \frac{1}{m^2}\sum_{j,j'} k(y_j, y_{j'})$$

, with $k(\cdot,\cdot)$ being any RBF kernel. Kernel is basically a similarity function that aims at computing the dot product between two vectors in a feature space where the data are linearly separable. The fascinating thing about kernels is that if we wanted to project our data in a high-dimensional space , this would be computationally infeasible but with kernels we do not have to use extensive computations as the domain knowledge of the feature space is contained in the kernel formula. A simple example of a kernel is $(x, x') = x^T x'$. In this particular algorithm, both Laplassian and Gaussian kernels were found to work well.

The interesting part of the experiments though, is the comparison between the previous algorithm presented, BCQ and some other algorithms like naive RL and behaviour cloning. The experiments were divided into 3 parts, our dataset D spawning from a random policy, from a mediocre scoring policy and from an optimal policy. The results are impressive, as BEAR consistently outperforms the other algorithms in the average quality data, which ofcourse are the most important category of data, as the most commonly available data fall into that category. In the random dataset, BEAR fares better than most of the other algorithms, except naive RL which sometimes gets on top, as every actions has the same probability. BCQ on random data resembles the behaviour policy so it fails to find the optimal actions and just behaves randomly, showcasing bad results. On the other hand, on data generated by an optimal policy, BEAR achieves similar results to BC, which is expected to do the best, as it mimics the actions of the optimal policy. BCQ also performs similar to BC, because it constrains its distribution to be close to the original optimal behaviour policy distribution.

4.3 Uncertainty based methods

While policy constraint methods provide a solution to the OOD action problem, there are also negatives associated with them. For example, the behaviour policy needs to be estimated from our static dataset in order to constrain our training policy, with wrong estimation of the behaviour policy leading to unexpectedly bad results. Another negative effect is that they are usually described as being too conservative in the sense that they resemble the behaviour policy in a large scale. For instance, if some arbitrary actions in the action space of the environment offer 0 reward, then we would not want to constrain the policy in these actions, but policy constraint methods blindly following the behaviour distribution would not treat these states as a special case. A way to deal with the problem of conservatism in the action space is to implement uncertainty based methods that introduce an uncertainty factor and are being divided into model-based and model-free.

4.3.1 Model-based uncertainty methods

We can extend the notion of offline RL into model-based offline RL by using our dataset and training NN's into modeling the environment. So, we train a probability transition model that given a state and an action, provides us with the next state and a reward function model that predicts the reward granted. Then we can perform planning given our handmade model of the world. But even in the process of planning, unseen outcomes will occur, leading into the OOD action problem. To solve this we can either use policy constraints as we've shown before or we can shape the parameterized reward function into being conservative. This means that in actions which we have encountered the reward function will remain consistent, whereas in unseen actions it will behave in a conservative way.

A recent example of such a method is Model-Based Offline Reinforcement Learning (MOReL). MOReL's main goal is the avoidance of over-estimating rewards that are involved in unexplored spaces of the environment, according to the dataset given. So, if the dataset does not span the entire action space then the model will be flawed. The way this is addressed in [13] is by dissecting the state-action space into known and unknown regions. Then whenever an unknown state-action pair is met, the reward function becomes conservative, while when a known pair is encountered then the reward function stays unchanged. This is achieved by using an uncertainty factor $u(s,a)$

$$r'(s,a) = r(s,a) - \lambda u(s,a)$$

This uncertainty term is linked with the estimation of the knowledge of a given state-action pair. The modelling of uncertainty is achieved by training an ensemble of models[13] and inspecting their accordance. The variance of OOD actions will be much higher than the variance of actions present in the dataset, amongst the models. This is used by model based algorithms in different ways.

First, [13] uses the disagreement notion, meaning that the difference between model predictions is dictating the value of $u(s,a)$. So, if the maximum distance between models in a given state-action pair is bigger than an arbitrary threshold, then the uncertainty factor is maximized. That leads to avoiding state-action pairs that cause the models to deviate from each other. On the other hand, Model-based Offline Policy Optimization (MOPO) takes a different approach. The dynamics of the environment are modelled by a neural network outputting a Gaussian distribution over the next state and reward and N dynamic models are being trained independently[14]. As [14] describes, model ensembles are used a lot in model-based RL because of their ability to capture the essence of the population mean and the variance of a Gaussian model can retrieve the uncertainty. So a solution to that is to set the uncertainty factor $u(s,a)$ equal to:

$$u(s,a) = \max_{\mathrm{i}=1}^{N} \left|\left|\Sigma_{\varphi}^{i}(s,a)\right|\right|_{F}$$

,where φ is the parameter of the variance in the neural network. The final formula using this method is :

$$r^{\sim}(s,a) = r'(s,a) - \lambda \max_{\mathrm{i}=1}^{N} \left|\left|\Sigma_{\varphi}^{i}(s,a)\right|\right|_{F} \qquad \boldsymbol{Eq\ 4.3}$$

,where $r'(s,a)$ is the mean reward predicted by the neural network. In both cases of MOReL and MOPO, after we obtain the reward function we can simply perform planning under our model.

Regarding the experimental results featured in both [13] and [14] we notice that model-based methods consistently outperform the policy constraint methods by averaging a larger amount of returns. As we discussed earlier, that is completely expected for the reason that policy constraint methods exhibit too much conservatism even in unnecessary cases. The fact that interests us the most though, is the observations in the MOPO algorithm [14] in tasks that require OOD generalization. Specifically, a SAC is trained for two environments and the rewards handed to the trajectories are altered from the original rewards in order to incentivize the agent to achieve something else. So, the agent now has to perform a different task. The results show that model-based algorithms like MOPO are significantly outperforming model-free methods and are able to generalize given the need for an OOD policy.

4.3.2 Model-free uncertainty methods

Generalization is a very important concept in offline RL, as it is a requirement for the solid behaviour of the trained policy compared to the behaviour policy of the static dataset. So, in order to achieve a different , in a better way, policy we really need our model to be able to generalize. In [15] two deep Q-learning algorithms are presented, Ensemble-DQN and REM, which use ensembles in order to attain generalization.

Ensemble-DQN is what its name suggests and basically is an ensemble of Q-functions that are approximated by a neural network. So, it resembles DQN but does take into account the expectation of multiple TD learning errors, averaging them. To be more precise to formulate the error at the k_{th} Q-function we use:

$$\Delta_\theta^k(s,a,r,s') = Q_\theta^k(s,a) - r - \gamma \max_{a'} Q_{\theta'}^k(s',a')$$

As we observe, each error is the TD learning update between our estimate Q_θ and the target Q-value. So, the way the loss function is modelled in the algorithm is:

$$L(\theta) = \frac{1}{K}\sum_{k=1}^{K} E_{s,a,r,s'\sim D}[l_\lambda\,(\Delta_\theta^k(s,a,r,s'))] \qquad \textbf{\textit{Eq}}\ \mathbf{4.4}$$

, and it is the average of the expectation of each Q-function passed on to l_λ which is the Huber loss[23] defined as :

$$l_\lambda(u) = \begin{cases} \frac{1}{2}u^2, & if\ \ |u| \leq \lambda \\ \lambda\left(|u| - \frac{1}{2}\lambda\right), & otherwise \end{cases}$$

REM however, deploys an ensemble of Q-functions like ensemble-DQN with the main difference being that these Q-functions are mixed at the end and the TD update takes into account the ensemble of Q-functions as a single Q-function. More specifically:

$$\Delta_\theta^\alpha(s,a,r,s') = \sum_k \alpha_k Q_\theta^k(s,a) - r - \gamma \max_{a'} \sum_k \alpha_k Q_{\theta'}^k(s',a')$$

If you pay close attention here you will notice that α is different than the a denoted as action. That is because α is a coefficient of each Q-function with the sum of them being equal to 1, i.e a convex combination. These coefficients stem from a probability distribution that is arbitrarily selected and is chosen to be a uniform distribution that's normalized in order to become a valid categorical distribution [16]. So, the previous equation of ensemble-DQN, becomes:

$$L(\theta) = E_{s,a,r,s'\sim D}\left[E_{\alpha\sim P_\Delta}\left[l_\lambda\left(\Delta_\theta^\alpha(s,a,r,s')\right)\right]\right] \qquad \boldsymbol{Eq\ 4.5}$$

,with the main difference being that the coefficients are chosen from the distribution P_Δ which we described earlier and the loss function not being averaged, since the error term Δ_θ^α already contains the sums of the Q-functions

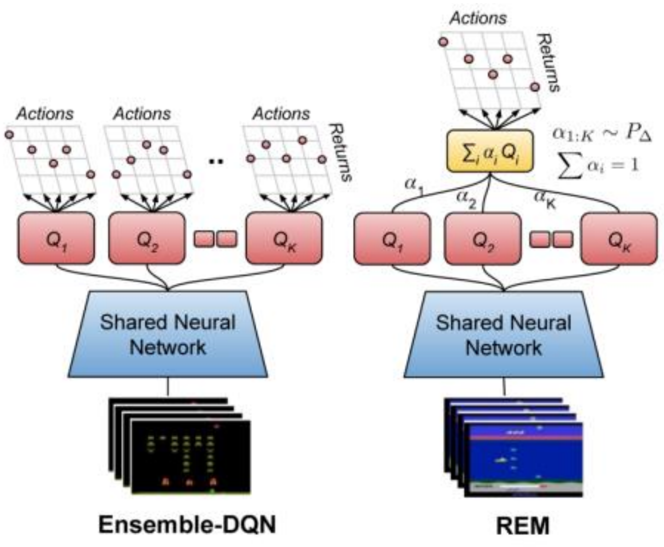


Figure 4.2 : Ensemble-DQN and REM architectures[15].

Experiments conducted by the authors of [16] show that in the offline case REM performs better than standard DQN and ensemble-DQN, because of the noise of random ensembles leading to more robustness, whereas the ensemble-DQN just averages the values. While in the paper they don't consider uncertainty , REM in particular works well in high coverage datasets for the reason that it grasps the essence of the environment by the random ensembles that it deploys. That also provides a boost to the generalization problem because as we covered earlier, the abundance of Q-function estimates renders the algorithm able to draw better conclusions on its predictions.

4.4 Policy evaluation methods

All the methods that were described previously regarding to offline RL, were attempting to discover the most optimal policy that would characterize the environment. But what if we want to calculate the value function of a given policy without running simulations of that policy in the environment? That's where the offline policy evaluation methods come into play.

4.4.1 Importance sampling (IS)

The objective of RL is the expected return following a given policy:

$$J(\pi) = E_{\tau \sim p_\pi(\tau)}[\sum_{t=0}^{T} \gamma^t r(s_t, a_t)]$$

But the issue arising here is that we are not able to sample from policy , so we have to acquire samples from the behaviour policy π_β in an offline setting. This can be attained by using importance sampling which follows the methodology below:

$$E_\pi[X] = \sum_{x\in X} x\pi(x) = \sum_{x\in X} x\pi(x)\left(\frac{\pi_\beta(x)}{\pi_\beta(x)}\right) = \sum_{x\in X} x\rho(x)\pi_\beta(x), \quad where\ \rho(x) = \frac{\pi(x)}{\pi_\beta(x)}$$

$\rho(x)$ is called the importance sampling ratio and is the ratio between the policy that we want to evaluate and the policy which we can sample upon. The previous equation is approximately equal to:

$$\frac{1}{n}\sum_{i=1}^{n} x_i p(x_i)$$

as:

$$E[X] \approx \frac{1}{N}\sum_{i=1}^{N} x_i$$

with the samples x_i being drawn from our distribution $\pi_\beta(x)$. So, the previous objective function that was induced can be transformed with importance sampling into:

$$J(\pi) = E_{\tau\sim\pi_{\beta(\tau)}}\left[\frac{\pi(\tau)}{\pi_{\beta(\tau)}}\sum_{t=0}^{T}\gamma^t r(s_t,a_t)\right] = E_{\tau\sim\pi_{\beta(\tau)}}\left[\prod_{t=0}^{T}\frac{\pi(a_t|s_t)}{\pi_\beta(a_t|s_t)}\sum_{t=0}^{T}\gamma^t r(s_t,a_t)\right] \quad \boldsymbol{Eq\ 4.6}$$

where

$$\frac{\pi(\tau)}{\pi_\beta(\tau)} = \frac{p(s_1)\prod_t p(s_{t+1}|s_t,a_t)\pi(a_t|s_t)}{p(s_1)\prod_t p(s_{t+1}|s_t,a_t)\pi_\beta(a_t|s_t)} = \frac{\pi(a_t|s_t)}{\pi_\beta(a_t|s_{t)}}$$

because the transition probabilities are the same for each policy and that's very convenient for us in a model-free environment. Lastly Equation 4.6 becomes :

$$J \approx \sum_{i=1}^{n} w_T^i \sum_{t=0}^{T}\gamma^t r_t^i \qquad \boldsymbol{Eq\ 4.7}$$

which stems from the importance sampling methodology with w_T^i being the average of the importance sampling ratio. A problem with this approach is the product of the policy probabilities is exponential and may attain high variance as they tend to zero. A way to get around this is using the Weighted Importance Sampling (WIS), which is a normalization of the Importance Sampling formula induced by dividing the return function $J(\pi)$ with the average cumulative reward importance ratio at horizon t regarding our dataset, ω_t [23][24] .Indeed:

$$\omega_t = \sum_{i=1}^{n} \frac{\rho_{1:t}^{i}}{n}, \qquad where\ \rho_t = \frac{\pi(\tau)}{\pi_\beta(\tau)}$$

And

$$J \approx \sum_{i=1}^{n} \frac{w_T^i}{\omega_t} \sum_{t=0}^{T} \gamma^t r_t^i \qquad \boldsymbol{Eq\ 4.8}$$

WIS greatly improves the variance of the estimator but introduces some bias, as each instance of ρ is divided by the weighted sum ω_t. Another detail that gives somewhat better results with lower variance than IS is the per-step importance sampling estimator, a modification that constrains the sampling ratio until timestep t, as we do not require future actions and states in order to update until that timestep[26]. So, with that in mind, Eq. 4.7 and 4.8 become:

$$J_{stepIS} = \frac{1}{n} \sum_{i=1}^{n} \sum_{t=0}^{T} w_t^i \gamma^t r_t^i \ and\ J_{stepWIS} = \frac{1}{n} \sum_{i=1}^{n} \sum_{t=0}^{T} \frac{w_t^i}{\omega(t)} \gamma^t r_t^i$$

Still after implementing these methods, variance is something that bothers us and an attempt to address this issue became reality with the Doubly Robust Estimator. The logic behind using the Doubly Robust Estimator is to achieve a balance between variance and bias in a standard environment. As Fig. 4.3 shows, high variance is described by having many predictions that fall short in achieving the true value, while some do but in general is something that we want to avoid as being unreliable. We can recall the MC methods as having too much variance, because we need to specify the full trajectory before updating, but the number of trajectories existing in the environment could be massive. Meanwhile, the bias is a systematic error of the true value that’s not big in numbers as some predictions in a case with high variance could be, but happens to almost every sample. The best match, of course is both the variance and bias being low which is presented in the top left of Fig 4.3. So, Doubly Robust Estimator is trying to achieve this particular case by deploying a model planning term which has high bias and low variance combined with a high variance importance sampling term. This methodology is well known in statistics and has been transferred into the field of Machine Learning and in our case Reinforcement Learning by [23][35].

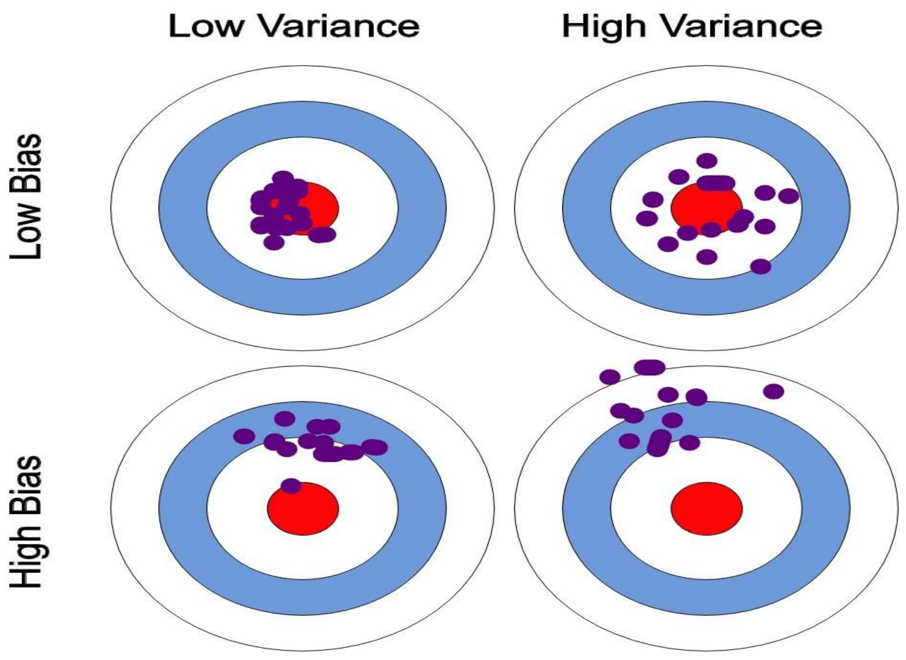


Figure 4.3 : Bias-variance tradeoff [35].

In order to derive the doubly robust estimator we need to implement the recursive form of Eq. 4.7 . We know that the value function in the form of Bellman equation is the product of the policy (in our case the importance sampling) with the rewards gotten in that timestep plus the value of the previous value function. In that case:

$$J_{step-IS}^{H+1-t} = \rho_t(r_t + \gamma J_{step-IS}^{H-t})$$

Since J is basically the value function V at that timestep, we observe that the expected value of the sum in the parentheses is the action value function Q. In that way we can form a subtraction using a baseline Q' function, which can be estimated via regression or by using a model-free algorithm or even by planning on an environment whose dynamics are known. The new value in the parentheses is $r_t + \gamma J_{step-IS}^{H-t} - Q'(s_t, a_t)$, which is a much smaller value than $r_t + \gamma J_{step-IS}^{H-t}$, especially if the estimated Q function is close to the value of the recursive term and this is performed as a means to reduce the variance like we did in chapter 3 using the advantage function. Also an estimate of the value function is added to that product to promote the lower of bias [25]. The final formula is :

$$J_{DR}^{H+1-t} = V'(s_t) + \rho_t \left(r_t + \gamma J_{step-IS}^{H-t} - Q'(s_t, a_t) \right) \boldsymbol{Eq\ 4.9}$$

There are many variations of Doubly Robust Importance Sampling apart from the standard case which was introduced in [36]. Some of them use the bias-variance tradeoff more efficiently in order to achieve a better balance, but the underlying notion is what we described above.

4.5 Value function regularization methods

We've previously encountered constraint methods that employ constraints in the policy in order to make the learned policy stay relevant with the behaviour policy provided. But what if we could interfere with the Q-functions and the Q-values which are causing the OOD action problem, directly?

4.5.1 Conservative Q-Learning (CQL)

Conservative Q-Learning is basically a regularizer that can be added in a standard actor-critic setting in order to lower bound the Q-values selected, preventing them from bootstrapping into infinity. As referred in [27] the way to handle this overestimation problem is via modifying the objective of the Q-function update, as simply subtracting a term, like a policy divergence term, from the standard Q-function update would also form a constraint like the ones we've encountered before. The regularization term are two and they intuitively make sense as the first one samples the biggest Q-values in the whole state-action space that are in general the values of OOD actions and it minimizes them. So, the peaks that form and would form even more with bootstrapping, are avoided with that minimization term. The other term only samples from actions that belong into our dataset in a way to promote them by increasing their value. This way, the algorithm works by simultaneously lowering the Q-values of actions that make the bootstrapping error explode and are generally not reliable into being picked and increasing the actions that we've seen in the dataset and lead to reliably good results.

Experiments conducted using the CQL method seem to perform better than other much used methods in the field both in the reward return section and in the overestimation of Q values section. Agents trained with CQL are able to score higher than other agents and also perform better in tasks that require generalization by stitching trajectories that are suboptimal to form a highly optimal one. Furthermore, analysis of Q-function estimation by these methods was conducted and showed that the CQL algorithm estimates the Q-values much better than the other methods and even better than policy constraint methods, while attaining better results than them. The greatest thing about CQL is that it's able to outperform state of the art algorithms and its simplistic usage[27].

CHAPTER 5

RESEARCH AND CONCLUSIONS

5.1 Offline Datasets and Research

This subchapter is mainly focused on datasets that provide research opportunities and useful insight that comes up with the extensive study of dataset influence into the general field of offline RL. Here we'll examine the latest news on the field regarding benchmark opportunities for researchers that do not possess the resources in order to conduct offline RL real world experiments.

A very interesting project for those interested in doing practical research in offline reinforcement learning is the D4RL, which is short for Datasets for Deep Data-Driven Reinforcement Learning. D4RL provides benchmark tasks and datasets publicly, in order for aspiring researchers to get a hands-on experience with offline RL, given that most online RL benchmarks (like MuJoCo) are not aimed towards the offline aspect of it. There are a couple of design variables that one has to take into account if he is to design a well defined offline RL task [31]. Some of the hindrances that are apparent in RL environments and we would like to maneuver offline RL towards tackling them are described below.

First, we have the case of narrow datasets where the majority of data in the distribution are close to the mean. That poses a challenge, because the offline agent might diverge or even train a worse policy than the behaviour policy provided. As we observed in Chapter 4, a way to get around distributions that are close together is to deploy a conservative algorithm, namely a policy constraint one that does not make the resulting policy wander far away from the behaviour one.

In addition, a significant issue that surfaces in RL as a whole, are environments that provide sparse rewards. For example, Montezuma's Revenge(a grand challenge in the RL community as it's known for being one of Atari's hardest to solve games) is a case of environment that proves this issue, as in the game you have to do alot of unrewarded work in order to acquire a key that gets you to the next level. So, the goal here is to plan the path towards obtaining the objective and finally getting the reward, while in the meantime the rewards that the agent is receiving are very few and sparse. In offline RL though, sparse rewards without exploration add on additional burden, as we have no way to explore the environment, as for instance the sophisticated algorithms that now solve the Montezuma's Revenge game.

Moreover, a classical problem in offline RL is associated with the data provided by the behaviour policy. There is a problem when the data is suboptimal mainly in the imitation learning cases where the agent blindly follows the demonstrator's action to the point where it will mimic him even if he acts suboptimally. But as we've seen some offline algorithms work quite well in these cases where the data come from suboptimal demonstrators, in the sense that they outperform the most standard algorithms like AC and DQN. The data, apart from being suboptimal, might also be logged. This means that they cannot be explicitly used for a particular task we want to accomplish, but may come in handy in the so called process of "stitching" [32]. For example, if a car is given the directions in the dataset to reach a certain destination from the starting point and then from the middle point to reach the final destination, it should learn to generalize and "stitch" these trajectories together by forming the path from the starting to the endpoint. Practically, in Figure 5.1 we observe the red line combining the two different trajectories.

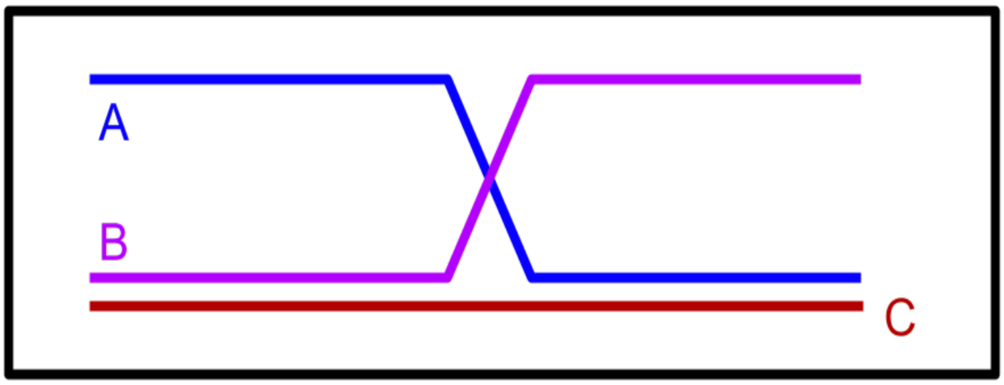


Figure 5.1 : Stitching of two different trajectories[32].

So, we've covered some of the basic problems that an offline agent will have to face if he is to succeed in offline RL training. With this in mind, the D4RL team created some tasks that utilize the previous issues we mentioned, providing the datasets which makes for a unique opportunity of being able to conduct research without requiring prohibiting amount of resources. The datasets provided consist of some mazes with the first one being a simple 2D maze but the more demanding one is the AntMaze, which employs an 8 degrees of freedom ant robot containing a much more complex morphology, which resembles the motions of a robot. These two tasks determine the agent's performance on stitching trajectories in an attempt to discover the shortest path between two nodes. MuJoCo and GymAI benchmarks are also included with added datasets for offline RL learning.

There is a variety of benchmarks being offered by the conductors of the project but they are mainly focused on the aspect of robotics, traffic control and autonomous driving[32]. They also add that important topics of research are not included in the benchmark environments, such as stochastic systems, mainly healthcare, financial markets and advertisement, which can pose a challenge into research. Furthermore, since the benchmark datasets are composed of simulated data, the desire to move into real world environments with real world data is apparent.

Another very interesting and innovative project is the RoboNet, a dataset that's specifically focused on the field of robotics[37]. The author's main concern was that in order to train robotic models effectively so they would be capable of generalizing, not only a sophisticated algorithm is required, but also the dataset should be large enough. This is required not only for generalization's sake but also for the need to pre-train an algorithm with a diverse dataset, before jumping into the training part with a smaller dataset[37]. Something along the lines of ImageNet dataset is what the authors dreamt, which is a database that accommodates around 15 million images. So, the inspiration came from ImageNet and its diversity, having a wide range of image classes is what motivated the creators of RoboNet into including a dataset with 7 different robots from 4 different research institutions. The whole dataset consists of approximately 162k trajectories that translate into 15 million frames, but the categories are broad and ranging from different robot types to different gripper types and even camera configurations and arena environments.

Apart from the description of the datasets, in [38] experiments are also conducted for the large scale data to be tested. The first step, is defining the task that is to be evaluated which is referred to as a relocation task. A relocation task involves moving an object that is not present in the dataset to an arbitrary location. The task is considered successful if the object that our agent moved, is covering most of the space of the location. After the task is defined, we move on to the next part which is a test of generalization. After the model has been trained with RoboNet, a test is conducted to observe whether it can generalize to new viewpoints. Two cases are compared, the case where a model has been trained on 90 different viewpoints and the case where a model has been trained on 1 viewpoint, different than the one that they are tested upon. The results show that the model that has been trained on the most viewpoints, relocates the object on a lesser distance from the goal than the other model, something which signifies the generalization that is achieved by being trained on multiple viewpoints. But the greatest challenge is referred to as the model being able to generalize in a setting with a previously unseen robot. Robots do not only vary in appearance but they also possess different dynamics [37] . This experiment is conducted following 3 separate training cases. The first case is initializing the weights randomly and training the model using data from the target robot and specifically 400 trajectories. The second case is to train the model using data from the target robot, but with more than 400 trajectories. The third case is pretraining on the RoboNet without using target robot trajectories and then fine-tuning using 300 to 400 trajectories of the target robot like before. The success rate of the third case is higher than the other 2, a result that praises the usefulness of training with a big and diverse dataset for better generalization, even if the dataset does not contain the target values. This experiment shows that the models pretrained on the RoboNet perform better than if they were trained from scratch with the target robot trajectories. A problem that's observed in the RoboNet dataset is underfitting, regarding models that are video predicting. So, the next step is to train two different models, one that has 200 million parameters and another one with 500 million parameters. The result they got is that logically, the model with the most parameters had the lowest error but still suffered from underfitting. To sum up regarding this paper, the authors after performing the experiments conclude that although the results were encouraging, the tasks were mostly trivial, with most of them requiring to pick an object and place it in the goal. The dataset is also something that worries them, because the policy used to generate the data in it, was a random policy so future work would require picking up a more sophisticated policy.

In this chapter we saw some of the datasets currently deployed in the field of offline RL. Regarding the D4RL project the benchmarks provided are quite sophisticated, with tasks requiring extensive generalization and could lead to significant breakthroughs through research, while the RoboNet project with its huge capacity of data could assist researchers into not having to spend valuable time and resources into generating their own data .

5.2 Conclusions

To conclude with, we covered some of the advancements happening in the field of Offline RL at this very instance, trying to solve the difficult problem of estimating a better policy than the one that we're provided with by the data. There are plenty of different methods as we've noticed in chapter 4, that are aiming towards correcting the OOD action overestimation and take better advantage of the behaviour policy. Policy constraint methods that endeavour to make the policy learned to resemble the behaviour policy in some way or the other. Uncertainty model based methods that deploy an ensemble of models into punishing unknown actions and choosing the known or model free methods that grind towards picking the best actions by estimating a plethora of Q functions. Importance sampling methods that utilize the samples of the known policy in order to evaluate the value function and also regularization methods that make use of regularizing terms in order to bound the Q-function into not exploding during backups. But there is also work to be done according to some open issues arising from these recent advances. In subchapter 5.1 we took a deep dive into the datasets that have been deployed recently and have unlimited potential regarding the research capabilities that they provide.

Each methodology that's being introduced in this paper has its own set of drawbacks and space for future corrections. For example, policy constraints, while possessing the capability of fixing the distributional shift problem, tend to rely too much upon the estimation of the behaviour policy[6]. Also the conservatism is an issue in the offline setting because we would never want to conserve the policy or Q-function if we were able to generalize well enough. So, there is a need for balance between the overestimation and underestimation of unseen actions. Also, importance sampling methods are comprised of high variance if the horizon in the current environment is too big and are in general too unreliable if our behaviour policy and our trained policy are different at large. Obviously, these algorithms suffer in the context of large state and action spaces and big horizon steps.

All in all, there’s too much research to be continued in offline RL, as significant progress has been made over the last two years, as a result of the apparent potential that the field possesses. With the improvement of datasets quality and the algorithms capabilities we can expect breakthroughs in automated driving, robotic control, decision making in healthcare and generally any activity that has a sequential form and does not allow for intensive interaction with the environment.

REFERENCES

[1] Gu, S., Lillicrap, T., Sutskever, I. & Levine, S.. (2016). Continuous Deep Q-Learning with Model-based Acceleration. *Proceedings of The 33rd International Conference on Machine Learning, in PMLR* 48:2829-2838

[2] Sutton, R., McAllester, D.A., Singh, S., & Mansour, Y. (1999). Policy Gradient Methods for Reinforcement Learning with Function Approximation. *NIPS*.

[3] Li, H., & Lau, T. (2019). Reinforcement Learning: Prediction, Control and Value Function Approximation. *ArXiv, abs/1908.10771*.

[4] Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction*. Cambridge, Mass: MIT Press.

[5] Kumar, A., Fu, J., Tucker, G., & Levine, S. (2019). Stabilizing Off-Policy Q-Learning via Bootstrapping Error Reduction. *NeurIPS*.

[6] Levine, S., Kumar, A., Tucker, G., & Fu, J. (2020). Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems. *ArXiv, abs/2005.01643*.

[7] Fujimoto, S., Meger, D., & Precup, D. (2019). Off-Policy Deep Reinforcement Learning without Exploration. *ICML*.

[8] Fujimoto, S., Hoof, H.V., & Meger, D. (2018). Addressing Function Approximation Error in Actor-Critic Methods. *ArXiv, abs/1802.09477*.

[9] Torabi, F., Warnell, G., & Stone, P. (2018). Behavioral Cloning from Observation. *ArXiv, abs/1805.01954*

[10] Kingma, D.P., & Welling, M. (2019). An Introduction to Variational Autoencoders. *Found. Trends Mach. Learn., 12*, 307-392.

[11] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition.

[12] Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller, M.A. (2014). Deterministic Policy Gradient Algorithms. *ICML*.

[13] Kidambi, R., Rajeswaran, A., Netrapalli, P., & Joachims, T. (2020). MOReL : Model-Based Offline Reinforcement Learning. *ArXiv, abs/2005.05951*.

[14] Yu, T., Thomas, G., Yu, L., Ermon, S., Zou, J., Levine, S., Finn, C., & Ma, T. (2020). MOPO: Model-based Offline Policy Optimization. *ArXiv, abs/2005.13239*.

[15] Agarwal, R., Schuurmans, D., & Norouzi, M. (2020). An Optimistic Perspective on Offline Reinforcement Learning. *ICML*.

[16] Tesauro, G. (1994). “TD-Gammon, a self-teaching backgammon program, achieves master-level play”. *Neural Computation, 6*(2), 215–219

[17] Bellman, R. (1958). Dynamic Programming and Stochastic Control Processes. *Inf. Control., 1*, 228-239.

[18] Witten, I. (1977). An Adaptive Optimal Controller for Discrete-Time Markov Environments. *Inf. Control., 34*, 286-295.

[19] Sutton, R., & Barto, A. (1981). Toward a modern theory of adaptive networks: expectation and prediction. *Psychological review, 88 2*, 135-70 .

[20] Klopf, A. (1997). Drive-reinforcement learning and hierarchical networks of control systems as models of nervous system function. *International Journal of Psychophysiology, 25*, 42-43.

[21] Klopf, A. (1988). A neuronal model of classical conditioning. *Psychobiology, 16*, 85-125.

[22] Watkins, C., & Dayan, P. (2004). Q-learning. *Machine Learning, 8*, 279-292.

[23] Huber, P. (1964). Robust Estimation of a Location Parameter. *Annals of Mathematical Statistics, 35*, 492-518.

[24] Jiang, N., & Li, L. (2016). Doubly Robust Off-policy Value Evaluation for Reinforcement Learning. *ICML*.

[25] Precup, D., Sutton, R., & Singh, S. (2000). Eligibility Traces for Off-Policy Policy Evaluation. *ICML*.

[26] Aviral , K.(2019, Dec 5). *Data-driven deep reinforcement learning*. BAIR. https://bair.berkeley.edu/blog/2019/12/05/bear/

[27] Kumar, A., Zhou, A., Tucker, G., & Levine, S. (2020). Conservative Q-Learning for Offline Reinforcement Learning. *ArXiv, abs/2006.04779*.

[28] Yamashita, R., Nishio, M., Do, R., & Togashi, K. (2018). Convolutional neural networks: an overview and application in radiology. *Insights into Imaging, 9*, 611 - 629.

[29] Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., & Kavukcuoglu, K. (2016). Asynchronous Methods for Deep Reinforcement Learning. *ArXiv, abs/1602.01783*.

[30] Wu Y., Mansimov E., Liao S., Radford A., Schulman J. (2017, Aug 18) . *OpenAI Baselines: ACKTR & A2C* . OpenAI. https://openai.com/blog/baselines-acktr-a2c/

[31] Fu, J., Kumar, A., Nachum, O., Tucker, G., & Levine, S. (2020). D4RL: Datasets for Deep Data-Driven Reinforcement Learning. *ArXiv, abs/2004.07219*.

[32] Tucker G.(2020, Aug 20). *Tackling Open Challenges in Offline Reinforcement Learning*. Google AI Blog. https://ai.googleblog.com/2020/08/tackling-open-challenges-in-offline.html

[33] *Soft Actor-Critic.* OpenAISpinningUp. Retrieved February 24, 2021 , from https://spinningup.openai.com/en/latest/algorithms/sac.html#exploration-vs-exploitation

[34] Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *ICML*.

[35] Yashwanth N. (2020, Sep 21) *What is Bias-Variance Tradeoff? Avoid the mistake of overfitting and underfitting*. Towardsdatascience. https://towardsdatascience.com/what-is-bias-variance-tradeoff-c8b19772e054

[36] Dudík, M., Langford, J., & Li, L. (2011). Doubly Robust Policy Evaluation and Learning. *ICML*.

[37] Dasari, S., Ebert, F., Tian, S., Nair, S., Bucher, B., Schmeckpeper, K., Singh, S., Levine, S., & Finn, C. (2019). RoboNet: Large-Scale Multi-Robot Learning. *ArXiv, abs/1910.11215*.

[38] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M.A. (2013). Playing Atari with Deep Reinforcement Learning. *ArXiv, abs/1312.5602*.

[39] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M.A., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature, 518*, 529-533.

[40] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review, 65 6*, 386-408

[41] Song, T. (2018, Feb 18). *Implementing the Perceptron algorithm from scratch with Python.* Tianyu Song. https://tianyusong.com/2018/02/23/implementing-the-perceptron-algorithm-from-scratch-with-python/

[42] Saha, S.(2018, Dec 15). *A comprehensive guide to Convolutional Neural Networks - the ELI5 way.* Towardsdatascience. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[43] Britz, D.(2015, Oct). *Recurrent Neural Networks, Introduction.* KDnuggets. https://www.kdnuggets.com/2015/10/recurrent-neural-networks-tutorial.html

[44] Seita, D. (2020, Jun 28). *Offline (Batch) Reinforcement Learning: A review of Literature and Applications.* Seita's Place. https://danieltakeshi.github.io/2020/06/28/offline-rl/

[45] Kumar, A., Levine S. (2020, Dec 12). *Offline Reinforcement Learning: From Algorithms to Practical Challenges.* Offline RL Tutorial- NeurIPS 2020. https://sites.google.com/view/offlinerltutorial-neurips2020/home