UNIVERSITÀ DEGLI STUDI DI TORINO

**Dipartimento Di Informatica**

**Corso di Laurea Magistrale in Informatica**


**Master's Degree Thesis**

# Overcoming the limits of Deep Reinforcement Learning with Model-Based approach

**Supervisor:**
**Prof. Marco Aldinucci**
**Antonio Mastropietro**

**Candidate:**
**Luca Sorrentino**

**Anno Accademico 2019/2020**

# Abstract

Reinforcement Learning (RL) is a subfield of Machine Learning where an agent learns to solve a task by interacting with the environment by trial and error without explicit knowledge. The agent receives a reward signal as feedback for every action it takes, and it learns to prefer those accompanied by a positive reward over those accompanied by a negative reward. This simple formulation allows the agent to directly choose the right actions from sensory observations, e.g. high dimensional inputs like camera frames, and to solve many complex tasks, like playing videogames or controlling robots.
The standard formulation exposed before is called Model-Free Reinforcement Learning because it does not require the agent to predict explicitly the environment dynamics, thus it can be viewed as a black-box approach. However, it requires a tremendous amount of experience and the lack of sample efficiency limits the usefulness of these algorithms in practice. One possible solution to overcome this problem is to combine the Reinforcement Learning framework with planning algorithms. This approach is called Model-Based RL. Instead of directly mapping observations to actions, Model-based RL allows the agent to plan explicitly the sequence of actions to be taken by observing the environment dynamics predicted by an environment model.
In recent years RL has been combined with Deep Learning algorithms to obtain outstanding results and reach superhuman performance in complex tasks. This combination of Reinforcement Learning and Deep Learning has been called Deep Reinforcement Learning (DRL). In this thesis, one of the state-of-the-art Model-Based DRL algorithms called PlaNet is deeply investigated and compared with the model-free DRL algorithm called Deep Deterministic Policy Gradient (DDPG). All the experiments are based on Deepmind Control Suite that is a set of continuous control tasks that are built for benchmarking reinforcement learning agents. Both the algorithms examined were tested on a subset of four environments. The main strengths and weaknesses of both approaches are highlighted in order to show if and how much a Model-Based RL can overcome the limits of Model-Free RL.

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement.

# Italian abstract

Il Reinforcement Learning (RL) è una tecnica di Machine Learning in cui vi è un agente che impara a risolvere un task interagendo col l'ambiente in cui si trova e di cui non ha nessuna conoscenza procedendo con un approccio trial and error. L'agente riceve un segnale di feedback chiamato reward per ogni azione che compie e impara a favorire quelle azioni accompagnate da un reward positivo a discapito di quelle accompagnate da un reward negativo. Questa semplice formulazione permette all'agente di prevedere le migliori azioni a partire dai sensori di input, come ad esempio i frame della camera, e di risolvere quindi molti task complessi, come giocare a videogiochi o controllare robot. La formulazione standard espressa finora viene definita Model-Free Reinforcement Learning perchè non richiede all'agente di costruirsi un modello dell'environment e di prevederne esplicitamente le dinamiche. Purtroppo, questo approccio diretto richiede un numero tremendamente elevato di esperienza e questo scarso livello di sample-efficiency limita l'utilità di questi algoritmi nell'uso pratico. Una possibile soluzione per superare questo problema è quella di combinare il Reinforcement Learning con algoritmi di planning. Questo approccio è chiamato Model-Based DRL. Invece di creare un mapping diretto tra osservazioni e azioni, il MBDRL permette all'agente di pianificare in modo esplicito la sequenza di azioni da intraprendere in base alle previsioni sugli sviluppi dell' environment.

Negli ultimi anni il RL è stato combinato con algoritmi di Deep Learning ottenendo cosi risultati eccezionali arrivando a superare umani esperti anche in task complessi. Questa combinazione di RL e DL prende il nome di Deep Reinforcement Learning (DRL). In questa tesi, uno degli più recenti algoritmi di Model-Based DRL chiamato PlaNet viene esplorato e comparato con un algoritmo di Model-Free DRL chiamato Deep Deterministic Policy Gradient. Tutti gli esperimenti sono basati sulla Deepmind Control Suite, un set di task creati per effettuare benchmark di agenti addestrati tramite DRL. Entrambi gli algoritmi presi in esame sono stati testati su quattro environments. I maggiori punti di forza e di debolezza dei due approcci vengono messi in luce per mostrare se e quanto l'approccio Model-Based sia in grado di superare i limiti del Model-Free DRL.

"Dichiaro che il sottoscritto nonché autore del documento è il responsabile del suo contenuto, e per le parti tratte da altri lavori, queste vengono espressamentedichiarate citando le fonti"

"When I started working on neural network models in the 1970s, people in artificial intelligence kept telling me that Minsky and Papert have proved that these models were no good."

Geoffrey Hinton

"I wanted to just prove everybody wrong."

Marshall Mathers

# Acknowledgements

Questo lavoro segna una tappa importante di un percorso iniziato già qualche anno fa quando, implementando A* vidi per la prima volta un software che portava a termine un compito senza richiedere una esplicita soluzione da programmare. A* aveva però molti limiti, uno in particolare era la richiesta di un umano che si specializzasse abbastanza da riuscire a trovare un'euristica per quel problema. Da quel momento nacque in me la volontà di creare un agente che fosse il più autonomo possibile, limitando in maniere sempre maggiore l'ingerenza umana. Ben presto iniziai a concentrare le mie energie sull'apprendimento per rinforzo, per permettere alla IA di imparare direttamente dalla propria esperienza, e sulle reti neurali per aumentare la potenza espressiva della mente dell'agente (percezione visiva, memoria, capacità rappresentativa).

Quando ho iniziato questo percorso ero convinto che avrei saltato questa pagina, visto che ero circondato da gente scettica che non mi era di aiuto e anzi frenava il mio entusiasmo. Dopo poco fortunatamente trovai una via di fuga attraverso il web. Per cui il mio primo ringraziamento va a chiunque utilizzi questo potente mezzo per trasmettere conoscenza. In particolar modo vorrei ringraziare: Salman Khan, Andrew Ng, David Silver, Sergey Levine.

Dopo un primo anno di studio autonomo e parallelo rispetto a quello universitario, avevo maturato una piccola conscenza di base che ho poi avuto modo di ampliare durante il mio percorso in Addfor. Il mio secondo ringraziamento va quindi a loro per avermi accolto e per aver creduto ed investito in me, fornendomi l'attrezzatura tecnica necessaria ed il supporto, sia sul piano tecnico che su quello umano. Un ringraziamento particolare ad Antonio, Sonia ed Enrico.

Infine, il mio terzo ringraziamento va alla mia famiglia per il supporto, ai miei colleghi per avermi accompagnato in questo percorso ed ai miei amici (lontani e vicini) per avermi incoraggiato quando le aspettative si facevano vertiginose, per aver sopportato la mia assenza quando le scadenze si avvicinavano e per avermi sostenuto e consigliato quando gli esperimenti fallivano. Un ringraziamento particolare va a Erica per essere riuscita a farmi ritrovare la serenità persa in questi ultimi anni.

# Contents

# Chapter 1

# Introduction

The idea of **Reinforcement Learning** (RL) has been known since **Turing**'s time. He talked about it in his article "Computing machinery and intelligence": "We normally associate punishments and rewards with the teaching process. Some simple child machines can be constructed or programmed on this sort of principle. The machine has to be so constructed that events that shortly preceded the occurrence of a punishment signal are unlikely to be repeated. In contrast, a reward signal increased the probability of repetition of the events, which led up to it.[1]" This idea is now called reinforcement learning and consists of training an agent to achieve a goal by interacting directly in an environment without prior knowledge. The agent receives positive or negative feedback for each action it takes and tries to accumulate positive rewards. From that time to our day, a lot of progress has been achieved. In recent years RL has been combined with Deep Learning algorithms to obtain outstanding results and reach superhuman performance in complex tasks, for example learning to play Go from scratch [2] or flying an acrobatic model helicopter [3].

Unfortunately, in order to reach some impressive results, these methods require a prohibitive amount of samples. For example, in 2019, Open AI released OpenAI Five: the first AI able to beat the world champions in an e-sports game called Dota 2 [4]. Despite the incredible result, the training cost was tremendous, the authors report that the OpenAI Five has experienced an average of 250 years of simulated experience per day. This is the reason why the main exciting results are obtained with agents that act in a virtual environment. Sometimes, these algorithms are also used to train a real world agent [5], but the training is mostly performed into a simulator. Build a simulator each time we need to train an agent for a task can be too expensive,for example think about a robot that learns to run, and it is an open research problem how to transfer the learned policy reliably to the real world. At the same time, the idea of performing millions of experiments with a physical robot in a reasonable amount of time and without hardware wear and tear is unrealistic. Alan Turing had already foreseen this problem, and in his article, he says: "The use of punishments and rewards can at best be a part of the teaching process... By the time a child has learned to repeat 'Casabianca' he would probably feel very sore indeed". One of the most promising solutions to improve the sample efficiency is the Model-

Based approach that combines the power of supervised learning, the reinforcement learning framework, and the planning algorithms. The key idea is to learn the environment's transition model to allow the agent to simulate interactions instead of acting directly without any knowledge. There are several approaches to learn predictive models of the dynamic environment using pixel information. If a sufficiently accurate model of the environment can be learned, then even a simple controller can be used to control a robot directly from camera images [6]. Another advantage of the model-based approach is that once the agent learns the model dynamics, it could quickly adapt without a fully retraining whenever the reward function was switched to assign a new task [7].

Unfortunately, the research in model-based RL is not been very standardized. The authors often use different environments (sometimes self-designed environments), and sometimes they do not publish their code. This thesis aims to create a fair comparison, built over standard benchmark environments the Deepmind Control Suite [8], between one of the most promising model-based algorithms called PlaNet [9] and one standard model-free called Deep Deterministic Policy Gradient [10].
Similar work is being done in 2019 by Tingwu Wang et al. [11]. They gather a wide collection of model-based reinforcement learning (MBRL) algorithms and propose over 18 benchmarking environments specially designed for MBRL and compare the results also with a model-free algorithms. All the tested algorithms in that research work in low dimension. In our work, we use the PlaNet algorithm that promises to perform well with high dimensional input. Moreover, since we focus on a single algorithm, we also implement some variants to improve its results on a benchmark environment.

Therefore, the following research questions are addressed:

- What are the pro and cons of the two approaches?
- What is the effective sample efficiency improvement when we use a model-based algorithm?
- How does the training time change?
- Does the model still achieve the same results?
- What improvements can we apply to the model-based algorithm?

### 1.0.1 Thesis Outline

Following this introductory chapter, the thesis branches between two main concepts, Deep Learning and Reinforcement Learning. Both have an extensive dedicated chapter. Then the RL theory branches again between Model-based and Model-free algorithms. In the dedicated chapters some architectures are presented. Lastly, we present our experiments and conclusion. Chapter 2 presents some main concepts

of the Deep Learning that will be necessary to do the experiments. Both supervised and unsupervised learning techniques are applied in later chapters. Chapter 3 introduces to classic Reinforcement Learning theory with an explanation of the mathematical preliminaries associated with it. An introduction to the difference between the model-free and model-based methods is introduced. Chapter 4 discusses some Deep Reinforcement Learning algorithms including the first one used for this thesis called Deep Determinitic Policy Gradient (DDPG). Chapter 5 provides a focus over the model-based DRL algorithms and it introduces the second main algorithm used for this thesis (PlaNet). Chapter 6 introduces the suite of benchmark environments, evaluates the performance of the two proposed method and discusses the results. Moreover, it introduces a new technique developed in this thesis to improve the sample efficiency of the model-based algorithm. Chapter 7 concludes gathered from the current endeavor and considerations for future work.

# Chapter 2

# Foundamentals of Machine Learning

In this section, we introduce the basic of Machine learning. In particular, we introduce the concepts of supervised learning, unsupervised learning, self-supervised learning and reinforcement leaning. We explain the difference between them and present some algorithms used for the thesis. The reinforcement learning section is intended as introduction, while it will be deeply exposed in the next chapters. For further details, refer to the body of Deep Learning[12]

## 2.1 Introduction

In its famous book, Tom Mitchell [13] provided widely quoted definitions of machine learning. It says: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E. For example, a computer program that learns to play checkers might improve its **performance** as measured by its ability to win at the class of **tasks** involving playing checkers games, through **experience** obtained by playing games against itself."

Usually, machine learning algorithms are used for tasks that are too difficult to solve with fixed programs written by humans. There are many classes of tasks that can be approached with machine learning, for example classification, regression, clustering, dimensionality reduction, data generation, machine translation, anomaly detection, denoising, density estimation.

For every task, there is a specific metric. The metric is a quantitative measure of the learner's abilities to solve the task, even for data that it has not seen before. Informally speaking, there are two main metrics. The first one is the *loss* that measures the model's error. For example for the classification task we could use a *0-1 loss* that increment the total error rate by 0 when the input is correct classified and by 1 if it is none. The third is the *reward* that is a feedback value that an agent receive every time it takes an action. It is positive when the action is correct and negative otherwise. So the performances of the agent depend on how much reward it is able to collect.

Machine learning algorithms can be classified by what kind of experience they are allowed to have in three main categories:

- **Supervised learning**: if all the model experience is concentrated in form of a given dataset in which every example is expressed by a vector of features.
- **Unsupervised learning**: where the examples in the dataset are not labelled.
- **Reinforcement learning**: when the agent collect autonomously the experience in the dataset by interacting with the environment.

## 2.2 Supervised Learning

The problem that are faced with supervised learning are:

- **Classification**: The classification problem consists of approximating a function $f : \mathbb{R}^n \rightarrow \{1, ..., k\}$ where k is the number of classes in which the dataset is divided. This function $y = f(x)$ will map every vector of features in which the input is represented to the corrected categories identified by category $y$.
- **Regression**: In the regression task the program is asked to predict a continuous value relate to the input. So, in order to solve this task, the program should learn a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$.

It is time now to introduce a metric to evaluate algorithm performance. The task defines this metric. For example, the loss function that we use for the classification task is the **cross entropy loss**. This metrics measures how the predicted distribution is close to the one that we are trying to approximate. For the regression problem instead, we could use the *mean squared error* that increment the loss value proportionally to the distance between the correct answer and the model given value.

Once the model is trained, we want to know how it works with never seen before data to determine if the model has been capable of generalizing over the dataset. This ability is called **generalization**.

To do that, we divide the dataset into three parts:

- Train set: used for the training;
- Validation set: used to evaluate the model and tune the parameters before a new training iteration;
- Test set: used to calculate the final performance measure of the model;

It is essential to specify that the model does not see examples from the training's development and test set.

### 2.2.1 Recurrent Neural Networks

In this section, we briefly introduce the Recurrent Neural Network; for an extended explanation, see [14]. This part is also based on the Understanding LSTM Networks blog post [15].

These models are called "neural networks" because they are inspired by neuroscience. As the biological version, the artificial neuron is modeled as a central nucleus connected with different inputs. An artificial neuron's mathematical model is a weighted sum of the neuron connection $w_i$ and the respective input $x_i$.

$$a = w_1x_1 + w_2x_1 + w_3x_3 + w_ix_i + \ldots + b = \sum_{i=0}^{i=N} (w_ix_i) + b$$

We can rewrite more compactly the above formula using the matrix formulation: $a = Wx + b$ . In order to produce the neuron output, the calculated weighted sum is passed through a non-linear **activation function** $\Phi$. In that way the model acquire the ability to deal with more complex data respect to the one affordable by linear models.

$$o = \Phi(a) = \Phi(\mathbf{W}\mathbf{x} + b)$$

The early artificial neural model adopted the Sigmoid function as an activation function, but today the common choice is the Rectified Linear unit (**ReLu**). Connecting various neurons allow the model to increase the representation power. In the neural networks, each neuron receives inputs from the others and uses it to calculate its activation value and propagate it. This network is divided into layers, each one that contains many neurons. This architecture is called **Multilayer Perceptrons** (MLPs) or Feedforward network.

In the MLP, the first layer is called **input layer** and receive the input. The last one is called **output layer** and produces the output. All the intermediate layers are called **hidden layers**. Feedforward neural network is a universal function approximation. It defines a mapping $y = f(x;\theta)$ and learns approximate whatever function by merely learning the best set of parameter $\theta$. These models can learn a hierarchical representation of the data, so they do not need hand-engineered features. The training algorithm must use those hidden layers to produce the desired output, but the dataset does not contain explicit information on how to do that.

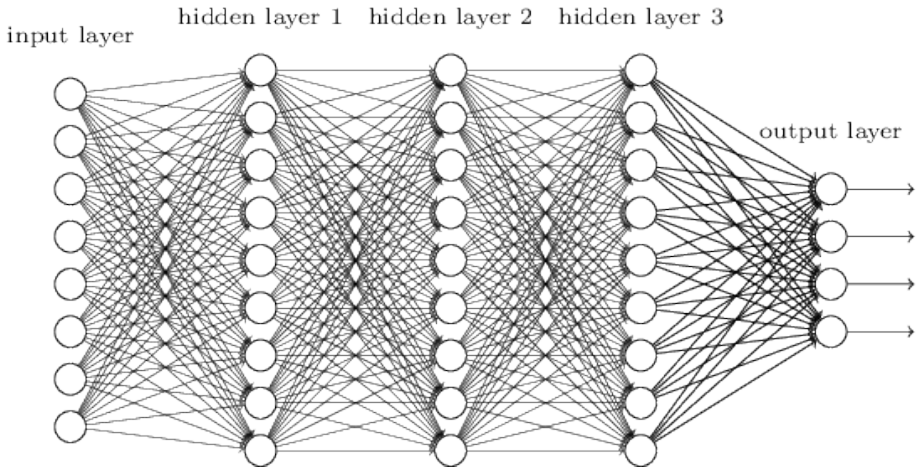



Figure 2.1: Fully-Connected Feed-Forward Network. Image from [16].

Neural network weights are trained with an algorithm called **Backpropagation**. The backpropagation algorithm is divided into two phases: a forward-pass and a

backward-pass. In the forward pass, the input is propagated throughout every layer and activation until the network's final output is computed. Then the error is calculated, calculating the difference between the network prediction and the training label. In the backward-pass, this error is backpropagated through all the layers to update each neuron's weights. In the last layer, the update is obtained, calculating the error gradient to understand the change rate of the layer weights. For the hidden layers, instead, the chain rule is applied to propagate the gradient by decomposing the derivative of the produced error recursively with respect to the parameters of the previous layers.

The standard MLP model has some limitations. After each processed example, the full state of the network is lost. As long as the input data maintains temporal independence between them, there are no problems for learning. Sometimes instead, the data is correlated, like with video frames or words in a sentence. In that case, we need a model that can face these correlations even without knowing the input sequence length. We can extend the "feedforward network," in which the path of the information strictly goes from input to the output by adding a feedback connection in which the information can go back to the model. This extension allows us to introduce the notion of time to the model. This new model is called a recurrent neural network (**RNN**). In a RNN any state depend from both the current input and the network state in the previous time step. Lastly, even if the expressive power grows exponentially, both the inference and training grow quadratically, and they are also differentiable end to end, so they are trainable with the backpropagation algorithm. It is time now to introduce some details of the RNN. We define a sequence of data as a arrays of data points $x^{(t)}$ extracted from a discrete sequence of *time steps*, each one indexed by t and is expressed with real-valued vectors. Both the input and the target are represented by a sequence $(x^{(1)}, x^{(2)}, ..., x^{(T)})$.

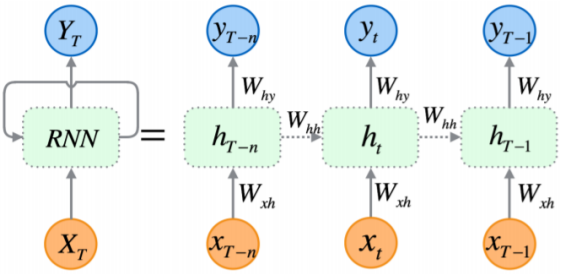



Figure 2.2: Standard RNN architecture and an unfolded structure with T time steps. Image from [17].

For each time step $t$ the current node receive information from both the data point $x^t$ in input and the previous state $h^{(t-1)}$ from the hidden node. For each time step $t$, the current node receives information from both the input $x^t$ and the previous state $h^{(t-1)}$ and uses this information to update the current state $h^{(t)}$.

$$h^{(t)} = \sigma\left(W^{\text{xh}} x^{(t)} + W^{\text{hh}} h^{(t-1)} + b_h\right)$$

This current hidden state $h^{(t)}$ can be used to calculate the current output $y^{(t)}$.

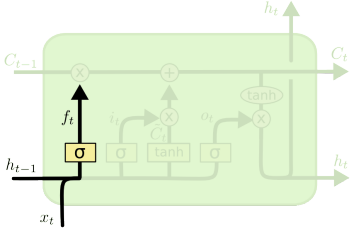
$$\hat{y}^{(t)} = \text{softmax}\left(W^{\text{hy}} h^{(t)} + b_y\right)$$

The $W_{hx}$ is the matrix representing the weights from the input and the hidden layer. The $W_{hh}$ is the matrix representing the recurrent weight between the same hidden layer through different time steps. During the training, the error signal can be back-propagated through the entire unfolded network across all the time steps. The back-propagation algorithm, used in a context where time is involved, is called *backpropagation through time* (BPTT). When we try to backpropagate the error across many time steps we can easily came across a problem of *gradient exploding* or *gradient vanishing*. One possible solution is to limits the maximum number of time steps in which the error can be backpropagated. This solution is called Truncated backpropagation through time (TBTT). Another solution is to design a particular architecture to limit the vanishing gradient problem without sacrifice the ability to learn long-range dependencies. This second approach led to a new neural network architecture called *Long Short Term Memory* (**LSTM**)

### Long Short Term Memory

In 1997 Hochreiter and Schmidhuber presented a new RNN architecture called Long Short-Term Memory (LSTM). In this network, they introduce the memory cell, a new computational unit that replaces the traditional nodes in the network's hidden layers, to handle the vanishing gradient problem of the RNN. In the traditional RNN, the long memory is maintained through the weights that capture the general knowledge about the data. The short memory, instead, is represented by the activation function between each successive node. In the LSTM, the memory cell works like intermediate storage and replaces short and long-term memory. A memory cell is a composite unit that contains several gates that add or remove information to the cell state. A gate is a sigmoidal unit that multiplies its output, between 0 and 1, with the value of another node to decide how much information can pass through. We can describe the works of the LSTM through a sequence of 3 steps:

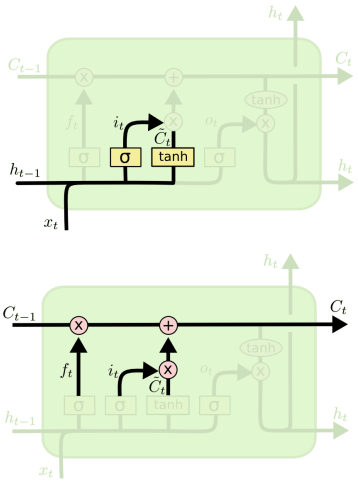
1. **The Filter**: in the first step, the LSTM decides which information to accept as input and which to forget. To do that it uses the *forget gate* that take in input the current input $x^{(t)}$ and the previous hidden layer $h^{(t-1)}$ and return a vector that will be used later to update the cell state.




$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figure 2.3: The output of the sigmoid is a vector of values from zero (completely forget) to one (completely keep). Image from [15].

2. **The Update**: in the second step, the LSTM decides how much information store in the cell state. This step is divided into two parts: in the first one, it uses the *input gate* to decide what value to update, and a tanh layer is used to create a vector of values called *candidate values* that could be added to the state. In the second one, the LSTM updates its internal state, called *Cell State*.



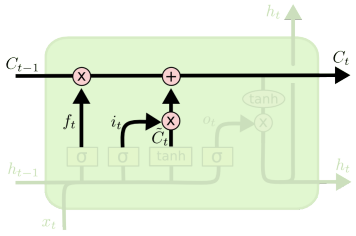

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$




$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 2.4: The Sigmod layer is used to decide what value to update, the Tanh layer to generate the vector of "candidate values" that could be added to the state. Next decides wich new information ignores, then in the tanh layer, it processes the new information respect the previously hidden layer and with the update gate decides which one to exclude for the update and which ti keep. Now it combines all this information to calculate the new Cell State. Image from [15].

3. **Compute Output**: in the final step, the LSTM combine previously hidden value, current input, and current cell state to calculate the new hidden value (that can be viewed as the current network's output). In this step, it is used the *output layer* to decide what part of the state can be outputted.




$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 2.5: Use the internal state and the output gate to produce the new hidden state. Image from [15].

The LSTM is still used since they have shown a better ability to handle long-range dependencies with respect to the simple RNNs.

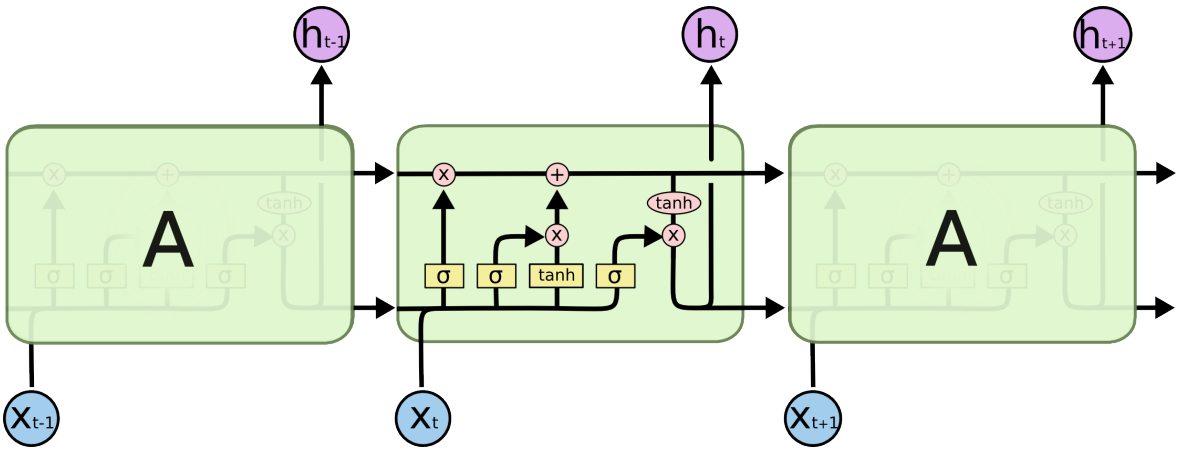



Figure 2.6: The LSTM architecture consist on a concatenation of LSTM cell units. Image from [15].

## 2.3 Unsupervised Learning

The principal tasks of unsupervised learning are:

- **Clustering**: is used to automatically dividing the dataset into clusters of a similar example.
- **Dimensionality Reduction**: reducing the number of observed random variables in a reduced set of principal variables.
- **Generative models**: learning a joint distribution over all the variables. In other words, a generative model simulates how the data is generated in the real world.

Recently a new subsets of Unsupervised learning methods has been introduce by Yann LeCun in and it is call **Self-supervised Learning**. According with Longlong Jing and Yingli Tian [18]

Self-supervised learning follow the same principle of provide to the learning algorithm a set of pair $X_i$ and $Y_i$ but with the difference that $Y_i$ are automatically generated without involving human annotations. That labels are called psuedo labels. We introduce one example of this technique: the **Variational Autoencoder**.

### 2.3.1 Variational Autoencoder

Variational Autoencoder is an example of the class of **generative model** methods. Here we provide a general explanation based on the introductory paper [19].

In a generative model setting, we try to estimate, explicitly or implicitly, a probability distribution of the data. Once the generative model has been tuned or trained, we can sample from the estimated distribution and we can generate new input data samples. The VAE is an implicit generative model, because it produces its own internal representation of the data without producing an explicit formula for the data

probability distribution. For this reason, this model is also used as a method to build a more compact representation of the data, minimizing the information loss. For the experiments in chapter 6, the VAE is used both as a dimensionality reduction algorithm and a generative model.

The VAE represents the marginal distribution over the samples with a function $p_\theta(x|z)$ parameterized with $\theta$. This probability is conditionally dependent on the latent variable z.

$$p_\theta(x) = \int p_\theta(x,z)dz = \int p_\theta(x|z)p(z)dz.$$

If we are in a discrete case (so if $z$ is a discrete variable), and $p_\theta(x|z)$ is assumed to be a Gaussian distribution, then $p_\theta(x)$ is a Gaussian mixture distribution. Instead, if we are working with a continuous variable z, then $p_\theta(x)$ can be seen as an infinite Gaussian mixture distribution. In this work $z$ is a continuous vector described by a simple Gaussian distribution and $p_\theta(x|z)$ is a conditional Gaussian.

We need a function that takes all the data in training set as inputs and extract from them the parameters of the latent Gaussian. Then we can sample the latent vector z from this latent Gaussian. We call this function **encoder** and we formally define it as:

$$q_\phi(z|x) = \mathcal{N}(\mu_\phi(x), \sigma_\phi(x))$$

Then we need also another function that uses this latent vector z to retrieve x. We call this function **decoder** and we formally define it as:

$$p_\theta(x|z) = \mathcal{N}(\mu_\theta(z), \sigma_\theta(z))$$

In order to find the best parameters $\theta$ we would like to use the Maximum Likelihood principle that is:

$$\theta \leftarrow \arg\max_\theta \frac{1}{N}\sum_i \log\left(\int p_\theta(x_i|z)p(z)dz\right). \tag{2.1}$$

Unfortunately, this formula is completely intractable, because of the integral in $\log\ p(x_i) = \log \int_z p(x_i|z)\ p(x)$, so we need another way.

Before moving on we introduce two concepts that we will use later: entropy and the Kullback-Leibler divergence.

**Entropy**: introduced in 1948 by Claude Shannon, the entropy measures the level of uncertainty of a stochastic variable outcome. For example, an event that has the high probability of occurring, say 90%, do not give us much information, so it has low entropy. Instead if we A coin instead, in which all the events have the same probability, will have a high level of entropy. The formula of the entropy is:

$$\mathcal{H} = -E_{x\sim p(x)}\left[\log\ p(x)\right] = -\int_x p(x)\log\ p(x)dx$$

**KL-Divergence**: is a measure of how well a distribution Q approximates another probability P, or in other words, how much information it's lost if the distribution Q is used instead of P.

$$D_{KL}(P||Q) = E_{x\sim P}\left[\log\frac{P(x)}{Q(x)}\right]$$

- Notice that it is not a distance between two distribution because is not symmetric:
$$D_{KL}(P||Q) \neq D_{KL}(Q||P)$$
- Where P=Q the KL-divergence is 0:
$$log\frac{P}{Q} = log1 = 0$$
- The KL- divergence is always a positive number.

Now we can go back to the VAE. The encoder part is represented by a neural network with parameters $w$, trained to approximate the latent distribution $q_\phi$. In other words, this neural network will find, from all the training examples x, the respective Gaussian parameters for the latent vector. So, more formally:

$$q(z) = \hat{q}_w(z|x) \approx q_\phi(z|x)$$

As we have seen before in equation 2.1 it is not possible to use directly that formula. We need to find another way to define the log $p_\theta(x)$:

$$\log\ p_\theta(x) = \mathbb{E}_{q_\phi(z|x)}\left[\log\ p_\theta(x)\right]$$

Now we apply the Bayes's theorem:

$$p_\theta(x) = \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(z|x)} = \frac{p_\theta(x,z)}{p_\theta(z|x)}$$

and substitute the $p_\theta$(x):

$$\log\ p_\theta(x) = \mathbb{E}_{q_\phi(z|x)}\left[\log\left(\frac{p_\theta(x,z)}{p_\theta(z|x)}\right)\right]$$

now we multiply by a constant $q_\phi(z|x)$ (the distribution of the encoder):

$$\log\ p_\theta(x) = \mathbb{E}_{q_\phi(z|x)}\left[\log\left(\frac{p_\theta(x,z)}{q_\phi(z|x)}\,\frac{q_\phi(z|x)}{p_\theta(z|x)}\right)\right].$$

next we decompose the expected value:

$$\log\ p_\theta(x) = \mathbb{E}_{q_\phi(z|x)}\left[\log\left(\frac{p_\theta(x,z)}{q_\phi(z|x)}\right)\right] + \mathbb{E}_{q_\phi(z|x)}\left[\log\left(\frac{q_\phi(z|x)}{p_\theta(z|x)}\right)\right].$$

We can rewrite the second term as the KL-divergence between $q_\phi$ e $p_\theta$:

$$\mathbb{E}_{q_\phi(z|x)}\left[\log\left(\frac{q_\phi(z|x)}{p_\theta(z|x)}\right)\right] = D_{KL}\left(q_\phi(z|x)||p_\theta(z|x)\right).$$

Now we focus on the first term:

$$\mathbb{E}_{q_\phi(z|x)}\left[\log\left(\frac{p_\theta(x,z)}{q_\phi(z|x)}\right)\right]$$

we rewrite the joint probability as a conditional probability:

$$\mathbb{E}_{q_\phi(z|x)}\left[\log\left(\frac{p_\theta(x|z)p_\theta(z)}{q_\phi(z|x)}\right)\right]$$

we apply the property of logarithms:

$$\mathbb{E}_{q_\phi(z|x)}\left[\log\ p_\theta(x|z)\right] - \mathbb{E}_{q_\phi(z|x)}\log\left(\frac{q_\phi(z|x)}{p_\theta(z)}\right)$$

we rewrite the second element as a KL-divergence

$$\mathbb{E}_{q_\phi(z|x)}\left[log\ p_\theta(x|z)\right] - D_{KL}(q_\phi(z|x)||p_\theta(z))$$

Now we can rewrite the entire formula as:

$$\log_{p_\theta}(x) = \underbrace{\mathbb{E}_{q_\phi(z|x)}[\log\ p_\theta(x|z)]}_{\text{=Reconstruction error}} - \underbrace{D_{KL}(q_\phi(z|x)||p_\theta(z))}_{\text{=First regularization term}} + \underbrace{D_{KL}(q_\phi(z|x)||p_\theta(z|x))}_{\text{=Second regularization term}}$$

Let us now examine all the components of the formula one by one.

Let's start with the first regularization term. It represents the similarity between the encoder and the latent distribution. Since both are Gaussians it is possible to calculate the KL in a closed form.

$$KL(p||q) = \frac{1}{2}\left[log\frac{\Sigma_2}{\Sigma_1} - d + tr(\Sigma_2^{-1}\Sigma_1) + (\mu_2-\mu_1)^T\Sigma_2^{-1}(\mu_2-\mu_1)\right]$$

The second regularization term instead, represents the similarity between the encoder and the true posterior $p(z|x)$ but it is not possible to calculate it.

We know that the KL divergence is always positive, so we can ignore the second regularization term and obtain a computable lower bound of the log $p_\theta(x)$ that is called Expected Lower BOund (ELBO).

Lastly, we need to find a way to calculate the Reconstruction error term. This term is the decoder's contribution to the final result: the ability to rebuild x given the latent vector z. We can approximate it through the sampling and SGD optimization, but we need to introduce a little trick before, called "**the reparametrization trick**."

We divide the training time into two-phase: forward propagation and backpropagation.

In the forward phase, the encoder produces the parameters for the latent distribution from the given input. From this distribution, the latent vector is sampled and given to the decoder. The decoder uses this vector to recreate the original input.

The difference between the decoder's output and the original input is the error of the VAE, and it must be backpropagated for all the computational graph.

In the backpropagation phase, the error passes through the decoder but fails to reach the encoder. That is because the sampling of z is a non-continuous and a non-differentiable operation; it has no gradient. So we need to make deterministic the choice of z but to keep stochasticity by applying the reparametrization trick.

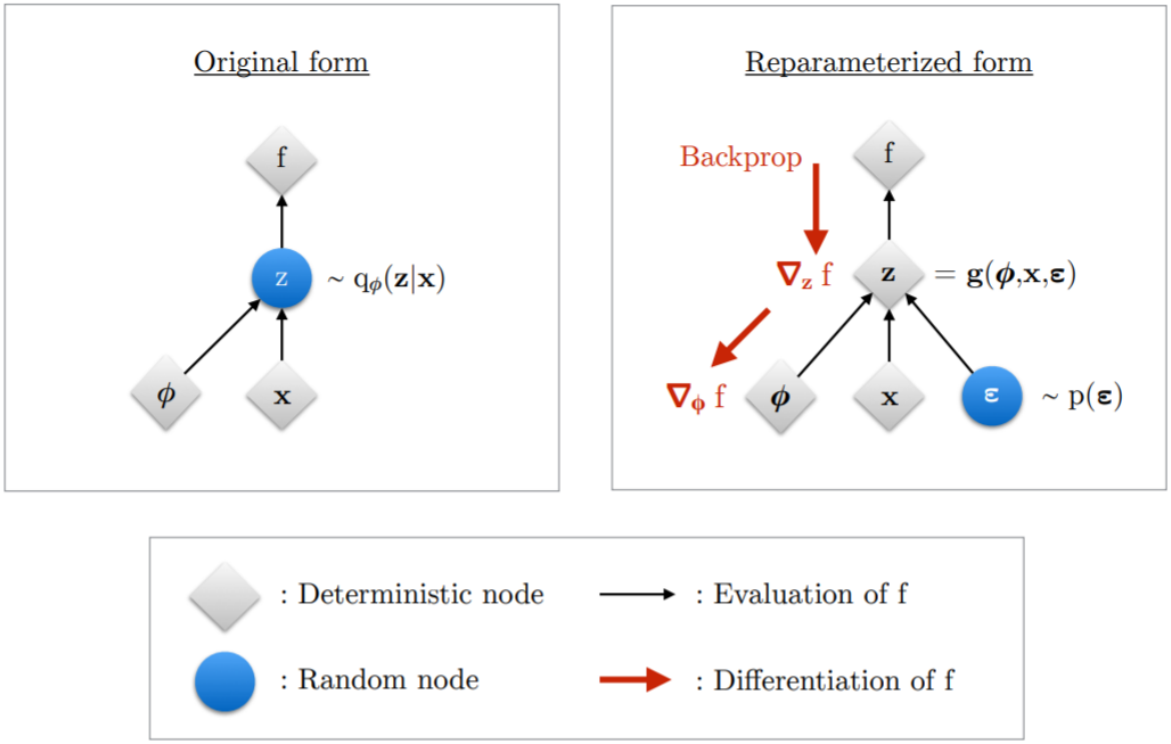



Figure 2.7: Illustration of the reparameterization trick.

This method consists of adding a new parameter called $\epsilon$ that is randomly sampled, but that is independent of encoder parameters $\phi$.

$$\epsilon \sim \mathcal{N}(0,1)$$

Since we cannot directly backpropagate the gradients through the vector z because of its randomness, we re-parameterizing this variable z as deterministic and differentiable.

Now the latent vector is no more sampled but computed:

$$z = \mu_\phi(x) + \epsilon\sigma_\phi(x)$$

The expectations can be rewritten in terms of $\epsilon$:

$$\mathrm{E}_{q_\phi(z|x)}[f(z)] = \mathrm{E}_{p(\epsilon)}[f(z)]$$

where $z = g(\epsilon, \phi, x)$. Finally it is possible to calculate $\nabla_\phi g(\phi, x, \epsilon)$

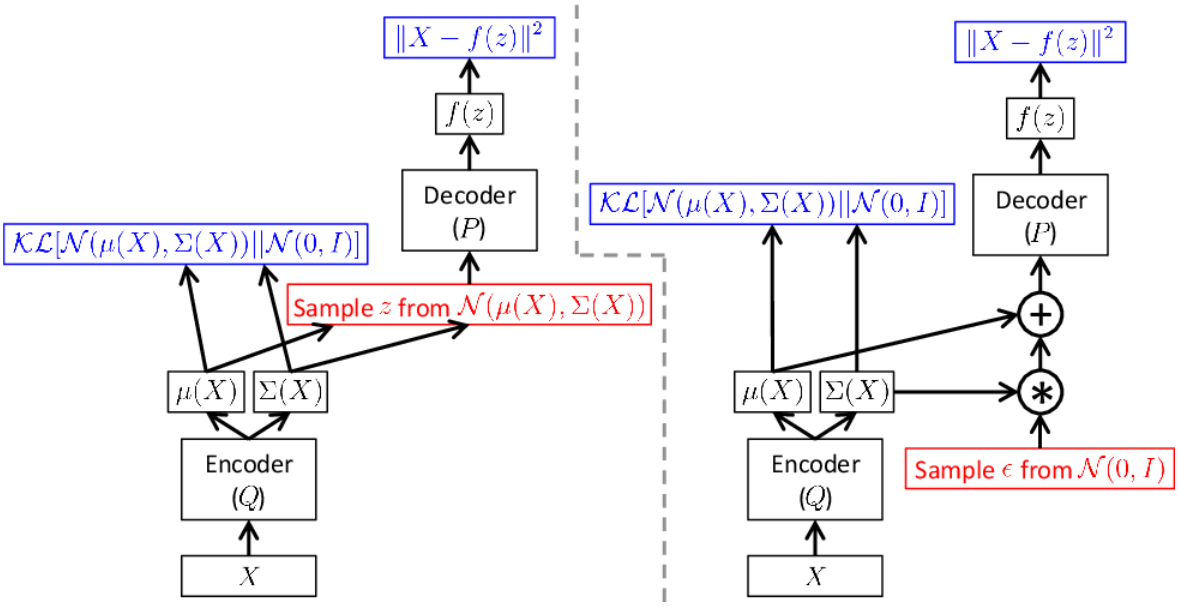



Figure 2.8: A training-time variational autoencoder implemented as a feedforward neural network, where P(X | z) is Gaussian. Left is without the "reparameterization trick", and right is with it. Image from [20].

## 2.4 Reinforcement Learning

In the reinforcement learning scenario the learner is an agent in an environment. The agent that does not know the environment dynamics, what its purpose is, and obviously how to achieve it. However, it can choose an action and perform it, and then it will receive a feedback signal that it will use to understand if it was a good or bad action.

There are infinite possible environments, and for each of them, there are infinite possible objectives. Therefore it is necessary to build a method capable of learning in a given environment without any supervision. The agent's objective function is independent of the environment and it is universal, so it is always the same. Each agent maximizes the feedback signal and indirectly, in doing so, solves the environment. The feedback signal is built in such a way that, maximizing it, will lead to resolving the environment.

Reinforcement learning also differs from unsupervised learning because while the first one is about to maximize a reward signal, the second one is about finding the hidden structure in the collection of unlabeled data. In the following chapter we explain in more detail the reinforcement learning theory with an explanation of the mathematical preliminaries associated with it.

# Chapter 3

# Elements of Reinforcement Learning

In this section, we introduce the basic theoretical concept of reinforcement learning and the mathematical preliminaries associated with it. In particular, we start from the formalism of MDP and then see as an example one of the first simple approaches to resolution. For further details, refer to chapter 3,4,5,6 of Reinforcement Learning: An Introduction [21]

## 3.1 Markov Decision Process

In the Reinforcement Learning context, there is an agent that learns how to achieve its goal directly by interaction with the environment.

**Markov Decision Processes** formally describe both the environment and the agent. To understand the MDP is necessary to introduce some concepts like *Stochastic Variable*, *Stochastic Process*, *Markov Chain*.

### 3.1.1 Markov Chain

We start introducing the Stochastic Variable and Stochastic Process.

**Stochastic Variable**: is a variable, usually expressed with a capital letter, whose possible values depend on a particular outcome of a random phenomenon. They are also known as Random Variables and, they can be *Discrete* or*Continuous*.

**Stochastic Process**: is a collection of discrete stochastic variables each one indexed by a value that represents a step of time in the process. This index is usually expressed like time step t.

So in a particular time step t, the process is in one of all its possible states mathematically expressed by a stochastic variable. More formally $s_t \in S$ where $S = \{s_0, s_1, ..., s_m\}$. The set of all possible states is called **State Space**.

The evolution of the system during the time is represented by a progression of the index from the current variable $s_t$ to the next one $s_{t+1}$.

The rule by which the index progress is called **Transition Function**, and it is not deterministic, so the process produces different sequences of states each time it is

run, even if it always starts from the same initial state. The value of a variable can influence that of the variables that follow it. Therefore the progression of the states depends on the whole sequence starting from the initial state. More formally this rule is expressed with a probability distribution:

$$P[S_{t+1}|S_1, ..., S_t]$$

A **Markov Chain** is a particular case of stochastic process that respects the Markov Property: "The future is independent of the past given the present".

This means that each state must capture all the relevant information from the environment at that moment, so when you have the state, you could throw away all the history. In other words, thanks to the Markov Assumption the transition function is conditionally independent from the past state if the current state are given.

More formally we say:

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

### 3.1.2 Markov Decision Process

A Markov Decision Process (MDP) is an extension of the *Markov Chain* that includes an agent that performs actions in the process, that in this context is called **Environment**.

The agent influences the evolution of the environment with its actions. So the state transition probability change including also the actions: $P(s, a, s^{'}) = [S_{t+1} = s'|S_t = s, A_t = a]$. The purpose of the agent is to led the evolution of the environment to a particular set of states, called **Goal States**. At every time step, the agent makes a decision on which is the best action to take. The process by which the agent chooses an action from the given state is represented by with a function, called **Policy Function**. It is represented by a probability distribution associated with every state in input. More formally we say: an agent follows the policy $\pi$ for every time step $t$ and for each $s_t \in \mathcal{S}$ it executes the action $a_t \in \mathcal{A}$ according to the probability $\pi(a|s)$. To guide the agent's decision process we assign a reward signal (a real number $r_t \in \mathbb{R}$) to every action that it executes. Having this reward signal we can say if a state is more desirable then one other.

The agent receives the initial state of the environment and uses it to choose an action. Then, the environment evolves its state into a new one depending on both the current state and the agent's action. At this point, the agent receives two inputs: the new state and the reward signal. It uses the reward signal to improve its own decision method and the new state to choose the next action.

This process is repeated iteratively until a final state is reached.

Every step of this process is called **Transition**: $s_t, a_t, r_t, s_{t+1}$. The list of all transition from initial state to the final one is called **Episode**.

There are some particular classes of MDP that are called **Infinite MDP** in which do not exist a final state, but the construction of the reward function is still possible.

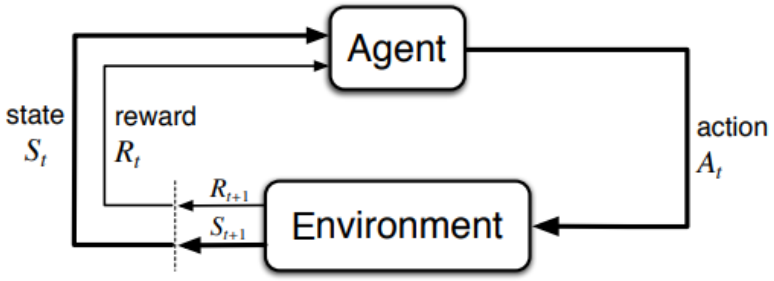



Figure 3.1: The agent-environment interaction in a Markov decision process. (Image source: Sec. 3.1 Sutton and Barto (2017) [21]

The agent can use the reward signal to know what is a good move and what it isn't. With this knowledge, it can learn the best strategies to achieve its goal. Assuming to be in a particular time step t of a finite MDP the definition of cumulative reward, from step t to the final step T, is:

$$G_t = R_t + R_{t+1} + R_{t+2} + ... + R_{t+T}$$

So the objective of the reinforcement learning is to find the parameters $\theta^*$ to the policy function $\pi^*$ that maximizes the expected cumulative reward of all the episodes.

$$\pi^* = argmax_\theta E_{\tau \sim \pi_\theta} \left[ \sum_t^T r(s_t, a_t) \right]$$

The reward value is often presented with the **Discount Factor**, generally expressed with the symbol $\gamma \in [0, 1]$. The purpose of the discount factor is to define the priority that the agent assigns to the future expected reward with respect to the immediate ones. So the formula of the discounted expected cumulative reward becomes:

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

Notice that the lower the gamma factor is and the more the agent will prefer the immediate reward than the long term values.

This value not only helps to represent the uncertainty about the future but it is also mathematically convenient because it can be used to end up with a finite number also in case of infinite sequence of states (using the sum of infinite series) if $\gamma < 1$. So it can be very useful for Infinite MDP.

To conclude a MDP is a tuple defined by $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \mathcal{Z}, \gamma >$:

- a finite set of states $\mathcal{S}$
- an initial state $s_0$
- a finite set of actions $\mathcal{A}$

- a state transition probability $P(s, a, s^{'}) = [S_{t+1} = s' | S_t = s, A_t = a]$
- $\mathcal{R}$ is a reward function, $\mathcal{R}(s, a)$
- $\gamma$ is a discount factor $\gamma \in [0, 1]$.

### 3.1.3 Partially Observable Markov Decision Process

A Partially Observable Markov Decision Process, also called POMDP, is a particular case of MDP in which the agent hasn't direct access to the full states but for each time step,it makes an observation that depends on it.

A POMDP in defined by a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \mathcal{Z}, \gamma >$, so respect to the classic MDP it required also:

- a finite set of observations O
- an observation function, $\mathcal{Z}(o, a, s^{'}) = P\left[O_{t+1} = o | S_{t+1} = s^{'}, A_t = a\right]$

In POMDP the transition is composed by the action, the reward and the observation of the environment. We define the entire sequence, from initial state to some time step t, as history $H_t$:

$$H_t = O_0, A_0, R_0, O_1, \ldots, O_{t-1}, A_{t-1}, R_{t-1}, O_t$$

During the history, the agent formulates hypothesis on what is the effective state behind each observation. To build this hypothesis, that is called *belief state*, it uses the history.

So, more formally, a belief state b(h) is a probability distribution over states, conditioned on the history h:

$$b(h) = \left(P\left[S_t = s^1 | H_t = h\right], ..., P\left[S_t = s^n | H_t = h\right]\right)$$

With the belief state the Markovian Assumption is no more valid.

## 3.2 Solving Markov Decision Process

In this section we introduce the two principal approach to solve an MDP, that are called *prediction problem*, used when a fixed policy is given and *control problem* used when there are no policy available.

### 3.2.1 Prediction Problem

The prediction problem consist of evaluating a given policy function in an unknown MDP. The metric used to evaluate a policy is the **value function**.

For every state the value function estimates how good it is for the agent that follows its given policy in the environment. This value is a scalar number and it is

expressed in terms of the expected cumulative discounted reward from the given state to the end. So, more formally:

$$v_\pi(s) = E_\pi \left[ G_t | S_t = s \right].$$

To explain how it works we first need to define a partial ordering over policies. One policy is better than another when it produces a greater value for each state. More formally:

$$\pi \geqslant \pi^{'} \forall s \in \mathcal{S} \iff v_\pi(s) \geqslant v'_\pi(s) \, \forall s \in \mathcal{S}$$

So also the definition of the best policy is related to the value function:

$$\pi_* \geqslant \pi \, \forall \pi \, (\forall s \in S) \iff v_*(s) \geqslant v_\pi(s) \, \forall s \in \mathcal{S}$$

From Sutton's book [21], in chapter 3.6, we can read: "there is always at least one policy that is better than or equal to all the other policies. This is an **optimal policy**." Starting from the definition of the value function, it is possible to derive the iterative formulation for any arbitrary state.

$$v_\pi(s) = E_\pi \left[ G_t | S_t = s \right] \tag{3.1}$$

$$v_\pi(s) = E_\pi \left[ R_t + \gamma G_{t+1} | S_t = s \right] \tag{3.2}$$

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s^{'}} \sum_r p(s^{'}, r|s, a) \left[ r + \gamma E_\pi \left[ G_{t+1} | S_{t+1} = s^{'} \right] \right] \tag{3.3}$$

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s^{'}, r} p(s^{'}, r|s, a) \left[ r + \gamma v_\pi(s^{'}) \right] \tag{3.4}$$

The equation 3.4 is the **Bellman equation** for $V_\pi(s)$ and it shows the relation between the value of one state and the value of its successor. "It states that the value of the start state must equal the (discounted) value of the expected next state, plus the reward expected along the way" [21]. From there, it is possible to derive the **Bellman Optimality Equation** used to calculate the optimal value function $v_*$. Note that this equation can be written independently to any particular policy. Intuitively, the best policy must suggest the action with the highest expected return from the given state. Therefore the optimal policy evaluation consist of finding the action that maximizes the value function of the successor state plus the expected reward obtained from the actual states to the next one s'.

$$v_*(s) = \max_a \sum_{s^{'}, r} p(s^{'}, r|s, a) \left[ r + \gamma v_*(s^{'}) \right]$$

### 3.2.2 Control Problem

The *control problem* consist of solving an MDP without a given policy. The metric used to build an effective policy is the **Q-Value function**.

The q-value, defined $q_\pi(s, a)$ is the expected discounted return after executing the action $a$ from $\pi(s|a)$ and then keeping to follow the actions from the policy $\pi$. It

is also called *action-value function*. More formally, we define the *action value function for the policy* $\pi$, $q_\pi$, as follows:

$$q_\pi(s,a) \doteq E_\pi \left[G_t | S_t = s, A_t = a\right] \tag{3.5}$$

$$q_\pi(s,a) \doteq E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} | S_t = s, A_t = a\right] \tag{3.6}$$

The optimal Q-Value can be defined as:

$$q^*(s,a) = \max_{a \in A(s)} q_{\pi^*}(s,a), \quad \forall s \in S, a \in A$$

It is also possible to write the $q_*$ in terms of $v_*$ as follows:

$$q_*(s,a) = E\left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\right] \tag{3.7}$$

From the last equation is possible to derive the **Bellman optimality equation** for $q_*$. Starting from $v*$:

$$\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_\pi * (s,a) \\
v_*(s) &= \max_a E_{\pi_*} \left[G_t | S_t = s, A_t = a\right] \\
v_*(s) &= \max_a E_{\pi_*} \left[R_t + \gamma G_{t+1} | S_t = s, A_t = a\right] \\
v_*(s) &= \max_a E_{\pi_*} \left[R_t + \gamma v(S_{t+1}) | S_t = s, A_t = a\right]
\end{aligned}$$

Now we can apply the previous formula:

$$q_*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a\right] \tag{3.8}$$

$$q_*(s,a) = \sum_{s',r} p(s', r | s, a) \left[r + \gamma \max_{a'} q_*(s', a')\right] \tag{3.9}$$

Starting from a given state, the agent must find the action that maximizes $q_*(s,a)$ without the knowledge of the possible successor states or the dynamics of the environment.

It's time to use all the information introduced in this chapter to show how to build the policy function. This method is called **Generalized Policy Iteration** (GPI) and it is the combination of two interacting processes called **Policy Evaluation** and **Policy Improvement**.

The first process makes the value function consistent with the current policy computing the $V_\pi$.

$$V_\pi(s) = E_\pi \left[r + \gamma V_\pi(s') | S_t = s\right]$$

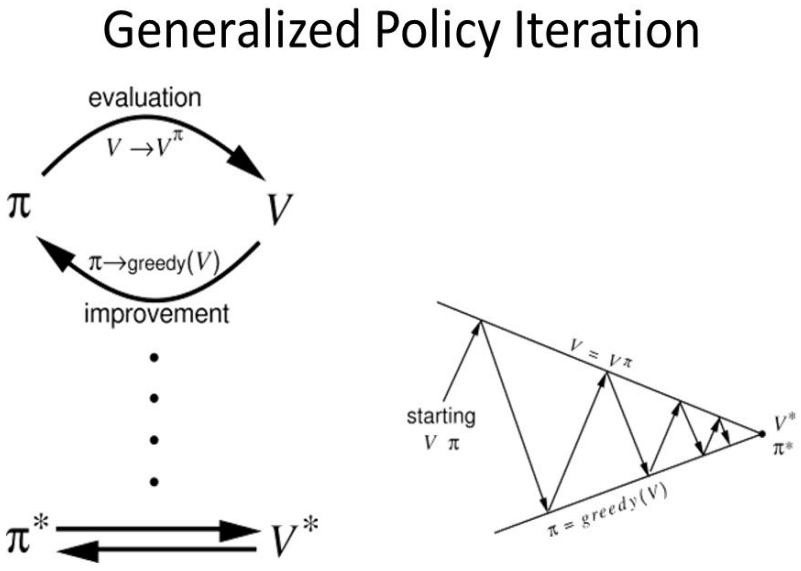



Figure 3.2: The GPI schema. (Image source: Sec. 4.6 Sutton and Barto (2017) [21]

The second process use the current value function to greedily improve the policy:

$$Q_\pi(s,a) = E\left[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s, A_t = a\right]$$

As said before, the GPI algorithm iterate over these two processes until it reaches convergence.

$$\pi_0 \xrightarrow{\text{evaluation}} V_{\pi_0} \xrightarrow{\text{improve}} \pi_1 \xrightarrow{\text{evaluation}} V_{\pi_1} \xrightarrow{\text{improve}} \pi_2 \xrightarrow{\text{evaluation}} \dots \xrightarrow{\text{improve}} \pi_* \xrightarrow{\text{evaluation}} V_*$$

## 3.3 Taxonomy of Reinforcement Learning Algorithms

Since the dynamics of the environment is not know, it is impossible to calculate directly the value or q-value function. For this reason there are two methods to approximate them, that are called **Monte Carlo Methods** (MC) and **Temporal Difference Methods** (TD).

Monte Carlo methods are based on the idea of repeated random sampling to estimate a distribution function. In this context, the function to approximate is the value functions need to the GPI schema explained in previous section. In order to compute the policy evaluation step, the agent performs several rollouts of the current policy accumulating the reward and the visited states of the entire episodes.

To accomplish the policy evaluation phase, the agent interacts with the environment accumulating experience. Every time it visits a state, it takes note of the number of times it encounters that state ($N_{(n)}$) and the cumulative reward obtained

in that episode from that state to the end ($C_{(n)}$).

$$N_{(s)} \leftarrow N_{(s)} + 1$$
$$C_{(s)} \leftarrow C_{(s)} + G_t$$

Now having the number of times the agent has visited a state and the cumulative total reward, it is possible to approximate the value function for each state by calculating the mean return.

$$V_{(s)} \leftarrow C_{(s)} / N_{(s)}$$

In the policy improvement step the agent chooses the action greedily with respect to the value function.

$$\pi^{'}(s) = \max_{a \in A} Q(s, a)$$

These two steps are iteratively repeated, and it can be shown that using the law of large numbers, it is possible to prove that the algorithm converges to the optimal policy and the optimal value function.

$$V_{(s)} \rightarrow v_\pi(s) \, as \, N(s) \rightarrow \infty$$

The problem with the Monte Carlo method is that it requires to finish the entire episode before to update the value function.

**Temporal Difference (TD)** methods are also based on the idea of GPI but differ from MC methods in the Policy Evaluation phase. Instead of getting to the end of the episode, these methods update the value function step by step. These methods use the current temporal estimates of the state value function, rather than relying on the complete return as the MC methods. This approach is called **bootstrapping**. To obtain a better approximation the algorithm recalculate the value of every state it visits and it adds to it the reward occured in that transiction, forming the so called **TD target**. The TD target is a slightly better approximation of the state value for that state, so the approximation must move in that direction. To move the approximation it's necessary to calculate di difference between the old estimation and the new one, producing the **TD error**. The entity of the update is controlled by a hyperparameter called learning rate $\alpha$. So, this processes is repeated at each time step, and the value function is continually update. This is called **online update**. The formula for the value function is:

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

The formula for the q-value function is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \tag{3.10}$$

These methods are referred to as **Tabular Methods** because the temporal results are cached in a table.

The most famous algorithm in this category is **Q-learning**. [22]. The formula used in this algorithm to update the Q-value estimate is the equation 3.8 that we

already presented before:

$$q_*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} q_*\left(S_{t+1}, a'\right) | S_t = s, A_t = a\right]$$

For the online update version of this formula the expectation is removed, so the formula became:

$$Q(s,a) = R(s,a) + \gamma\left(\max_{a'} Q(s',a')\right)$$

Applying this TD target to the generic version of the TD formula:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t))$$

The formula 3.10 the Q-learning formula add a *max* operation, this simplifies the algorithm approximating the optimal action-value function, $q_*$, directly.

**Algorithm 1** Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\text{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize $\mathcal{S}$
    Repeat (for each step of episode):
        Choose $\mathcal{A}$ from $\mathcal{S}$ using policy derived from **Q** (e.g., $\epsilon - greedy$)
        Take Action $\mathcal{A}$, observe $\mathcal{R}$, $\mathcal{S}'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q\left(S', a\right) - Q(S,A)\right]$
        $\mathcal{S} \leftarrow \mathcal{S}'$
    until $\mathcal{S}$ is terminal

Both the MC and TD methods are also **Model-Free algorithms** because they solve the MDP, calculating the value or q-value function, without knowing the environment dynamics.

However, an agent could also learn the transition probability function and the reward function and use them to accelerate the learning process or build a long term plan before acting. In that case, we talk about the **Model-Based algorithms** in which the model is learned, not given.

Another classification is based on how the methods find the best action to take. The option are **Value function based** or **Policy function based**. All the methods presented so far are value based: these methods learn the value function and then use it to choose the action with the best result. The policy function based, instead, search directly in the policy parameters space and ends up with the best policy.

Finally there is the **Actor-Critic** method that build both the value and the policy functions.

This method is composed by two element: the actor and the critic. The Actor is the Policy function and decides which action to take and the Critic is the Value function approximation and tells the actor how good its choice was and how to adjust it.

The critic updates value function parameters and the Actor update policy parameter using the value function suggested by the critic.

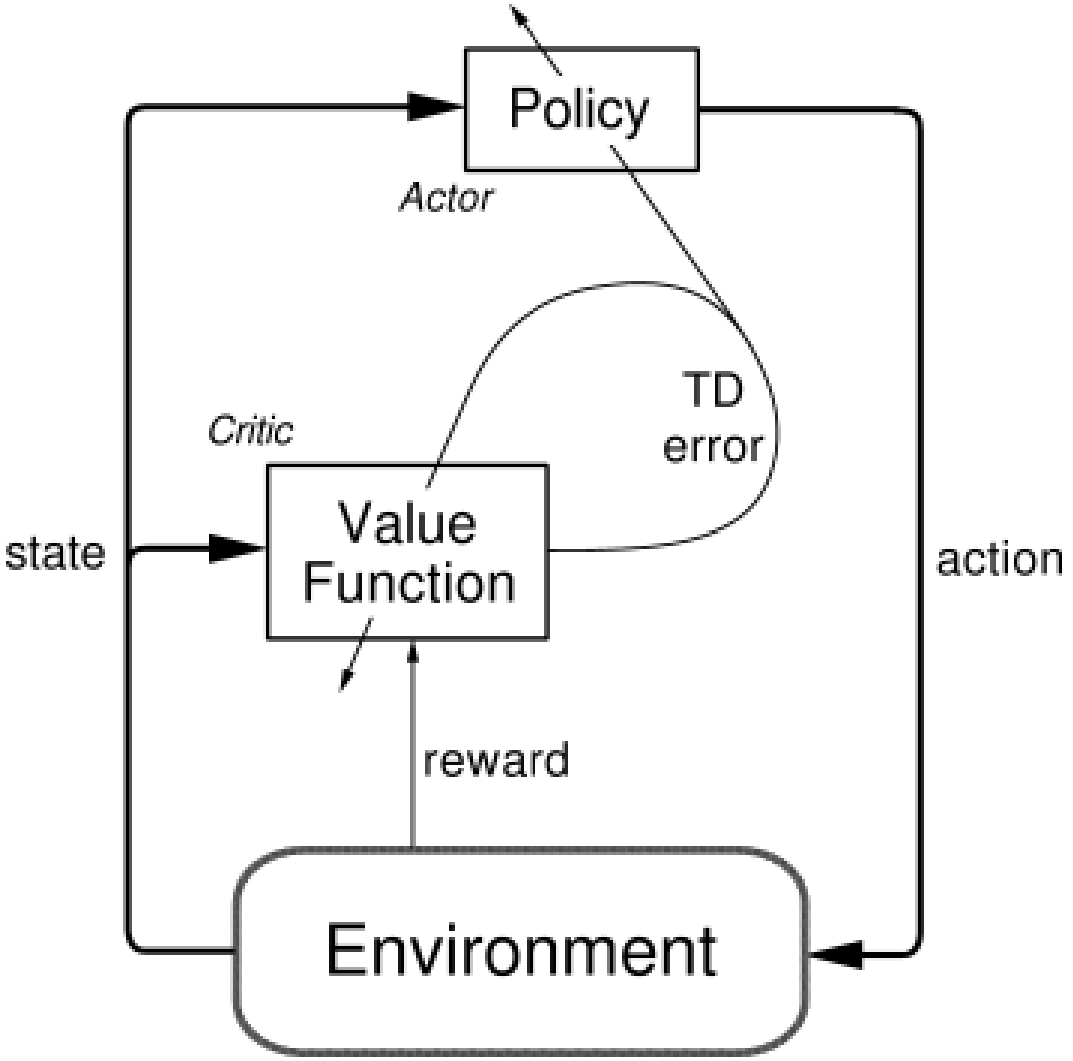



Figure 3.3: The actor-critic architecture. [Image source: Sec. 6.6 Sutton and Barto (2017) [21]]

In the next chapter, we will introduce an actor-critic method called DDPG that will be use it to do the experiments. In chapter 6, we compare the results obtained with DDPG and the results obtained with a model-based algorithm called PlaNet.

# Chapter 4

# Model Free Reinforcement Learning

## 4.1 Deep Reinforcement Learning

### 4.1.1 Deep Q Network

As we so in chapter 3, to solve an unknown MDP, we use the Bellman equation 3.7. We want to approximate the function of the optimal value iteratively until the algorithms converge, but to do this, we need to maintain the experience gained in the previous iterations. Originally, all the temporary q-value estimates were stored in a tabular, but this kind of solution worked well only with toy problems. The solution was the use of a function approximator instead of tabular methods. Initially, they tried whit a linear function approximation, but also that solution is not able to scale. The first, scalable and successfully, use of neural networks as a function approximator of Q-value was introduced in 2015 by Deepmind [23]. The solution proposed in this work is to build a Q-network and train it by adjusting the parameters $\theta$ to reduce the mean-squared error in the Bellman equation.

This network, called **Q-Network**, takes an observation as input and then give in output, with a single forward pass, the predicted Q-values for all possible actions.

Before this solution, the use of neural networks in the framework of reinforcement learning was known to be an unstable method. This instability has several causes: the correlation between transactions, the change in the distribution of data caused by the change in policy. To address these problems, they introduce two variants of the q-learning algorithm.

An **Experience Buffer Replay** was introduced to remove correlations in the observation sequence and smooth over changes in the data distribution.

So, at each time step $t$, the agent store its experience $e_t = (s_t, a_t, r_t, s_{t+1})$ in a dataset $D_t = e_1, ..., e_t$. At the learning time it the agent draw a random batch of experiences from the dataset and apply a Q-learning update.

The second variant is the use a second neural network called **Target Network** to perform a target estimate in the Q-update. So the Q-network is trained to reach the target network predictions that use an old set of weight. So, every C updates the Q-networks weights are cloned to generate a better set of weights for the Target-networks. This solution makes divergence or oscillations much more unlikely.

This algorithm was tested on a set of environments that replicated the games from an old console called atari Atari 2600. Reinforcement learning was usually applied to domains in which useful features were being crafted in low-dimensional state space. The DQN instead, was trained directly from high-dimensional inputs that were the raw frames of the games.

This choice led to a problem, one single frame does not contain enough information to be an effective state of an MDP since it violates the Markov property. In fact, having one single frame is not enough to predict the next one (for example, you cannot guess the position of an object in the next frame if you do not know its speed and direction). So, in that case, the environment is not an MDP but is a POMDP. To give to the network enough information they stacked more frames into one single input.

Using the frames as input allow the algorithm to be more general. In fact, DQN was able to achieve human-level performance over 49 different games using the same network architecture and hyperparameters.

**Algorithm 2** Deep Q-learning with Experience Replay

initialize replay memory $D$ to capacity N
initialize action-value function $Q$ with random weights $\theta$
initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
**for** episode=1,M **do**
  Inizialize sequence $s_1$=$x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  **for** $t = 1, T$ **do**
    With probability $\epsilon$ select a random action $a_t$
    otherwise select $a_t = argmax_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{t+1})$ from D
    **if** episode terminates at step j + 1 **then**
      set $y_j = r_j$
    **else**
      set $y_j = r_j + \gamma \, max'_a \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right)$
    **end if**
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j, \theta\right)\right)^2$
    with respect to the network parameters $\theta$
    Every C steps reset $\hat{Q} = Q$
  **end for**
**end for**

## 4.2 Policy Gradient

In chapter 3.3 we have introduced the difference between **Value function** based methods and **Policy function**. DQN is the most famous example of an algorithm

based on Value function. The Policy function methods instead, do no approximate a value function but learn directly the policy as $\pi(a|s;\theta)$. The objective is to maximize the expected reward cumulated during the episode. We now introduce **REINFORCE**, one of the algorithms based on this method, also called Monte-Carlo policy gradient.

### 4.2.1 REINFORCE algorithm

As we said we want to maximize the expected cumulative reward

$$\theta^* = \underset{\theta}{argmax} E_{\pi\theta}\left[\sum_t R(s_t, a_t)\right].$$

Since we work on a set of episodes, define $\tau$ as an episode and we set our objective function as the total reward accumulated over all the episodes:

$$J(\theta) = \sum_\tau \pi(\tau;\theta) R(\tau)$$

Every time that we change the parameters we move the distribution and so also the states that the agent visits. We need to find an objective that is independent from $\theta$ otherwise it is not possible to find the $\nabla\theta$ perform a gradient ascent step.

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_\tau \pi(\tau;\theta) R(\tau)$$
$$\nabla_\theta J(\theta) = \sum_\tau \nabla_\theta \pi(\tau;\theta) R(\tau)$$

Now we need to apply the likelihood ratio trick:

$$\frac{\nabla x}{x} = \nabla log x$$

So we first multiply $\nabla_\theta \pi(\tau;\theta)$ by the constant $\frac{\pi(\tau,\theta)}{\pi(\tau,\theta)}$ and then we can apply the trick:

$$\frac{\nabla_\theta \pi(\tau;\theta)\pi(\tau,\theta)}{\pi(\tau,\theta)} = \nabla(log(\tau;\theta))\pi(\tau,\theta)$$

Now we can rewrite the complete formula:

$$\nabla_\theta J(\theta) = \sum_\tau \pi(\tau,\theta)\nabla(log(\tau;\theta)) R(\tau)$$

We rewrite the formula with the Expected value form:

$$\nabla_\theta J(\theta) = E_\pi\left[\nabla_\theta(log\pi(\tau;\theta)) R(\tau)\right] \tag{4.1}$$

Since $R(\tau)$ is just a scalar representing the total reward collected over all the episodes, we now focus mainly on the log term, in order to understand how to calculate it. First, we examine the meaning of $\pi(\tau, \theta)$:

$$\pi(\tau, \theta) = p_\theta(s_1, a_1, ..., s_t, a_t)$$
$$p_\theta(s_1, a_1, ..., s_t, a_t) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

if we apply the logarithm to the policy probability we obtain:

$$log\, \pi_\theta(\tau) = log\, p(s_1) + \sum_{t=1}^{T} log\, \pi_\theta(a_t|s_t) + log\, p(s_{t+1}|s_t, a_t)$$

and now if we apply the gradient:

$$\nabla_\theta log\, \pi_\theta(\tau) = \nabla_\theta \left( log\, p(s_1) + \sum_{t=1}^{T} log\, \pi_\theta(a_t|s_t) + log\, p(s_{t+1}|s_t, a_t) \right)$$

Since we are looking for the gradient respect to $\theta$ we can eliminate all the terms the not depends on $\theta$.

$$\nabla_\theta log\, \pi_\theta(\tau) = \nabla_\theta \left( \sum_{t=1}^{T} log\, \pi_\theta(a_t|s_t) \right)$$

So if we put back $\nabla_\theta log\, \pi_\theta(\tau)$ to the objective 4.1 we obtain:

$$\nabla_\theta J(\theta) = E_\pi \left[ \nabla_\theta \left( \sum_{t=1}^{T} log\, \pi_\theta(a_t|s_t) \right) R(\tau) \right].$$

Lastly we can put the derivative inside the summation and rewrite the expected value:

$$\nabla_\theta J(\theta) = \frac{1}{n} \sum_{i=1}^{N} \left( \sum_{t=1}^{T} \nabla_\theta log\, \pi_\theta(a_t|s_t) \right) \left( \sum_{t=1}^{T} R(s_{i,t}, a_{i,t} \right).$$

So the final update rule formula is:

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

## 4.3 Actor-Critic:

So far we have seen a Value function method (DQN) and a Policy function method (REINFORCE). Reinforce is very unstable while DQN is not compatible with environments with continuous action space because of its max operator over all the possible moves for each step. So now we see an example of a new combination of the two, an algorithm that is based on Actor-Critic architecture from In chapter 3.3. This algorithm is called Deep Deterministic Policy Gradient (DDPG) [10].

### 4.3.1 Deep Deterministic Policy Gradient

Ddpg is an off-policy algorithm that can be used in continuous action spaces. The learning algorithm iterates between two phases: learning the Q-function from the data and use that value to learn a policy.

The Q-leaning phase: In this phase the objective is to approximate the Bellman equation:

$$Q^*(s,a) = \underset{s' \sim P}{\mathrm{E}} \left[ r(s,a) + \gamma \max_{a'} Q^* \left(s', a'\right) \right]$$

And we use the data collected during the training to approximate it. So having an a neural network $Q_\phi(s,a)$ with parameter $\phi$ as approximator and a buffer $D$ that contains all the transitions $(s,a,r,s',d)$ we can set up the **mean-squared Bellman error**.

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d) \sim \mathcal{D}}{\mathrm{E}} \left[ \left( Q_\phi(s,a) - \left( r + \gamma(1-d) \max_{a'} Q_\phi \left(s', a'\right) \right) \right)^2 \right]$$

As we already saw with dqn, also with ddpg a target network is involved to stabilize the training.

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d) \sim \mathcal{D}}{\mathrm{E}} \left[ \left( Q_\phi(s,a) - \left( r + \gamma(1-d) \max_{a'} Q_{\phi targ} \left(s', a'\right) \right) \right)^2 \right]$$

Now, how calculate $\max_{a'} Q_{\phi targ}\left(s', a'\right)$ if we are in a continuous action space environment?

The policy learning phase: Now we have a Q value $Q(s,a)$ and we want to find a deterministic policy $\mu_\theta(s)$ which gives the action a that maximize $Q_\phi\left(s,a\right)$.

But how to learn this policy? We know that the action space is continuous and we assume that the Q-function is differentiable with respect to action, so we can perform a gradient ascent step to find the best parameter to the policy.

$$\max_\theta \underset{s \sim \mathcal{D}}{\mathrm{E}} \left[ Q_\phi \left(s, \mu_\theta(s)\right) \right]$$

Because the policy is deterministic, during the training, we add some random Gaussian noise to let the agent to explore better the environment and to collect more varied data.

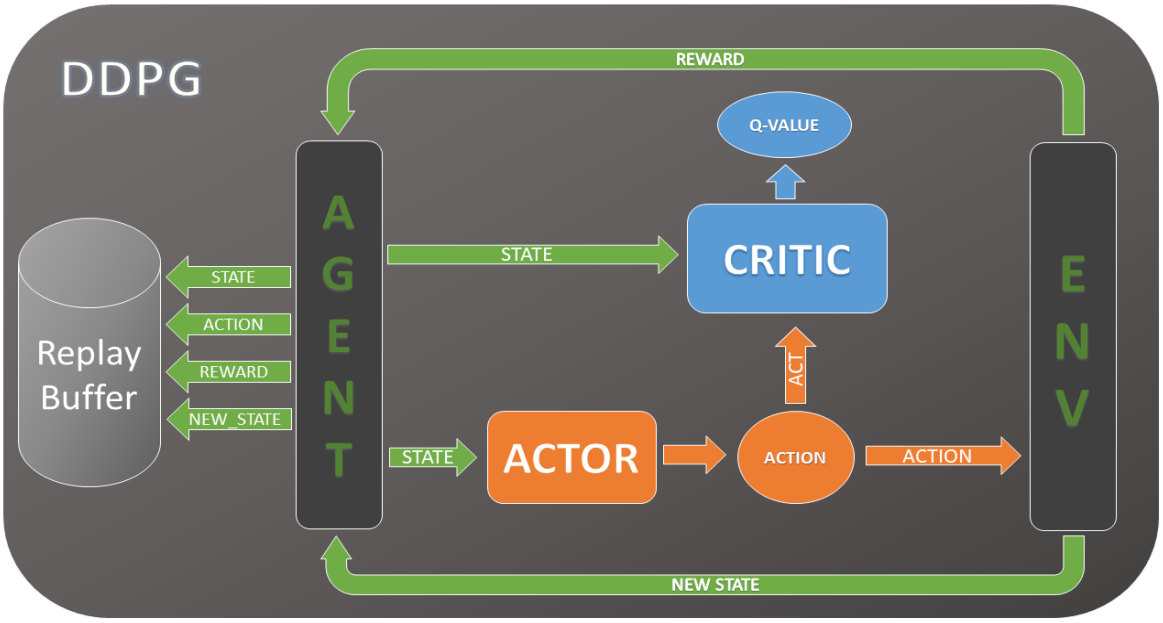



Figure 4.1: A visual representation of the DDPG architecture. The Q-values is used only at training time.

**Algorithm 3** Deep Deterministic Policy Gradient

Input: initial policy parameters $\theta$, Q-function parameters $\phi$, empty replay buffer D.
Set target parameters equal to main parameters $\theta_{targ} \leftarrow \theta$, $\phi_{targ} \leftarrow \phi$
**for** episode=1,M **do**
Observe state s and select action $a = \text{clip}\left(\mu_\theta(s) + \epsilon, a_{Low}, a_{High}\right)$, where $\epsilon \sim \mathcal{N}$
Execute $a$ in the environment
Observe next state s′,reward r, and done signal d to indicate whether s′ is terminal
Store (s,a,r,s′,d) in replay buffer D
**If** s′ is terminal state reset the environment state.
**if** it's time to update **then**
**for** however many updates **do**
Randomly sample a batch of transitions, $B = (s, a, r, s', d)$ from D
Compute targets
$y\left(r, s', d\right) = r + \gamma(1-d) Q_{\phi_{\text{targ}}}\left(s', \mu_{\theta_{\text{targ}}}\left(s'\right)\right)$
Update Q-function by one step of gradient descent using
$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B}\left(Q_\phi(s,a) - y\left(r, s', d\right)\right)^2$
update policy by one step of gradient ascent using:
$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi\left(s, \mu_\theta(s)\right)$
Update target networks with
$\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1-\rho)\phi$
$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1-\rho)\theta$
**end for**
**end if**
**end for**

# Chapter 5

# Model Based Reinforcement Learning

In this chapter we focus on Model Based approach, explaining how in works theoretically, why is useful and how could use it to create better agents. Then we present some significant proposal from the literature and lastly we introduce the model used for this thesis.

## 5.1 Model Based Reinforcement Learning

As we said in chapter 3 in the reinforcement setting, there is an environment in a specific state $s_t$ that receives an action $a_t$ from an agent. After receiving this action, the environment update its state using the transition probability function $s_{t+1} = f(s_t, a_t)$ and calculates also the correspective reward $r_{t+1} = r(s_t, a_t)$. The agent takes the new observation from the environment and uses that to choose the next action to take $a_t = \pi(a_t|o_t)$. Recall that in an MDP on observation correspond to the state, so $o_t = s_t$, while in an POMDP the observation is derivative of the state, so $o_t = o(s_t)$.

In the model-free setting, the agent will learn a policy that returns the best action directly to take in that state in order to maximize the expected cumulative reward. In the model-based setting, instead, the agent will learn to model the dynamics of the environment (forward model) by approximating the transition function and the reward function. So in case of MDP the model could be: $s_{t+1} = f_\theta(s_t, a_t)$. Instead if the environment is a POMDP the model need to use also the old observation and actions, in order to predict a new one. $o_{t+1} = f_\theta(o_0, a_0, ..., o_t, a_t)$

Once the agent is able to approximate the environment in its head, it is also able to simulate actions and predicts the possible consequences.

To be more specific, the agent plan a sequence of H (that stands for Horizon) actions $\{a_t, \ldots, a_{t+H}\}$, and then unroll the learned model H step into the future based on those actions. Now the agent can compute the objective function. $G(a_t, \ldots, a_{t+H}) = \mathbb{E}\left[\sum_{\tau=t}^{t+H} r(o_\tau, a_\tau)\right]$ to evaluate the current plan and performs some sort of optimization to find the best possible plan (often a genetic algorithm is used to this purpose) $a_t, \ldots, a_{t+H} = \arg\max G(a_t, \ldots, a_{t+H})$. This process is called **trajectory optimization**.

## 5.2 Planet

Planet is a new algorithm published in 2019 from the research team in Google AI [9]. In this paper, the agent is able to learn environment dynamics only through the observation and then can use this model to plan what action to take for each step. In order to achieve this task, the agent must solve three problems:

1. understanding the observation: capture the useful information contained in each frame and maintain them in memory
2. understanding the environment dynamics: be able to predict the next observation and the next reward having only the current observation and the current action as input
3. using its prediction to plan what action to take.

### 5.2.1 RSSM

Since the planning requires a considerable amount of predictions at every time step, the researchers decided to work in latent space. In other words, they do not use the entire frame to predict the next one, but they encode all the information in a vector obtained from neural networks, called latent vector. This advantage in terms of computational cost leads to a disadvantage for the agent that now has two jobs: first, it has to build a visual understanding of the environment and second, it has to find a way to solve the task. To be more specific, they use a convolutional neural network to capture all the spatial information from the image, and a GRU network (a simplified version of LSTM) to capture the temporal information across different time steps. Then they use both information to create the latent vector.

Now it is time to enter in technical details: We now considering sequences like $\{o_t, a_t, r_t\}_{t=1}^{T}$ where the index $t$ is used for the time step, $o_t$ is the environment observation for the current time step, $a_t$ and $r_t$ the current action and reward. The Planet model is composed of three sub models:

$$\begin{aligned} \text{Transition model: } s_t &\sim p\left(s_t \mid s_{t-1}, a_{t-1}\right) \\ \text{Observation model: } o_t &\sim p\left(o_t \mid s_t\right) \\ \text{Reward model: } r_t &\sim p\left(r_t \mid s_t\right) \end{aligned}$$

The transition model has the job of produce the current latent state by using the previous latent state and the current action. Then the observation model and the reward model will use it to reconstruct the observation and predict the reward obtained by the execution of $a_t$ in $s_t$.

The **observation model** is Gaussian with a mean parameterized by a deconvolutional neural network and identity covariance. The **reward model** is a scalar Gaussian with a mean parameterized by a feed-forward neural network and unit variance. In both cases, the loss is calculated through mean square error. The **transition model** can be viewed as a sequential VAE that is a convolutional variational autoencoder that receive in input an observation $o_t$ and an action $a_t$. The aim of the encoder

is to learn an approximation of the state posterior $q\left(s_{1:T} \mid o_{1:T}, a_{1:T}\right)$ from past observation and actions. This state posterior will contain all the useful information about the current state to allow the decoder (introduced above as the observation model) to uses this state $s_t$ to reconstruct the observation $o_t$ completely. When it produces the current posterior state, it needs to use also the information of the precedent state that is served as input in addition to the observation and action, and this is why it is called *recurrent* VAE. So the transition model approximate the true state posterior with $\prod_{t=1}^{T} q\left(s_t \mid s_{t-1}, a_{t-1}, o_t\right)$. We can find the true parameters of this Gaussian at training time because we have all the information necessary to calculate the loss value througth mean squared error. Another important point is that at training time we can always sample a batch of transitions from the experience replay and provide the current observation for each time step. At inference time instead, we only have the observation for the current step, but if we want to predict the posterior states of different steps in the future, we cannot provide the respective observation.

Intuitively if we ask the model to predict the next observations, it cannot require it as input. For this reason we use this model at training time to find the correct parameter of the posterior state and in inference time we use another model that not use the information about the current observation $o_t$ but it only require $s_{t-1}$ and $a_{t-1}$ $p(s_t|s_{t-1}, a_{t-1})$. This new model $p$ is trained to stay close to $q$ via kl-divergence. $KL\left[q\left(s_t \mid s_{t-1}, a_{t-1}, o_t\right) || p\left(s_t \mid s_{t-1}, a_{t-1}\right)\right]$.

Unfortunately, the only $s_{t-1}$ is not enough to maintain in memory all the useful information. The form of stochastic transition, in fact, not able to maintain information across multiple steps. For this reason, they also provide the model with a sequence of activation vectors $(h_t)_{t=1}^{T}$ from a GRU network. Combining these two methods, they create a new model called **Recurrent State-Space Model (RSSM)**. In RSSM, the internal state is composed of two parts: a stochastic one named $s_t$ (sampled from a Gaussian) and a determinist part $h_t$ (sampled from GRU). The final model is similar to the previous one:

$$\begin{aligned}
\text{Deterministic state model: } h_t &= f\left(h_{t-1}, s_{t-1}, a_{t-1}\right) \\
\text{Stocastic state model: } s_t &\sim p\left(s_t \mid h_t\right) * \\
\text{Observation model: } o_t &\sim p\left(o_t \mid h_t, s_t\right) \\
\text{Reward model: } r_t &\sim p\left(r_t \mid h_t, s_t\right)
\end{aligned}$$

*the information over action is already encoded in h

Now at inference time, the model can rely only on $h_t$.

Before moving on to the next paragraph, I left a quick visual recap of the Planet model.

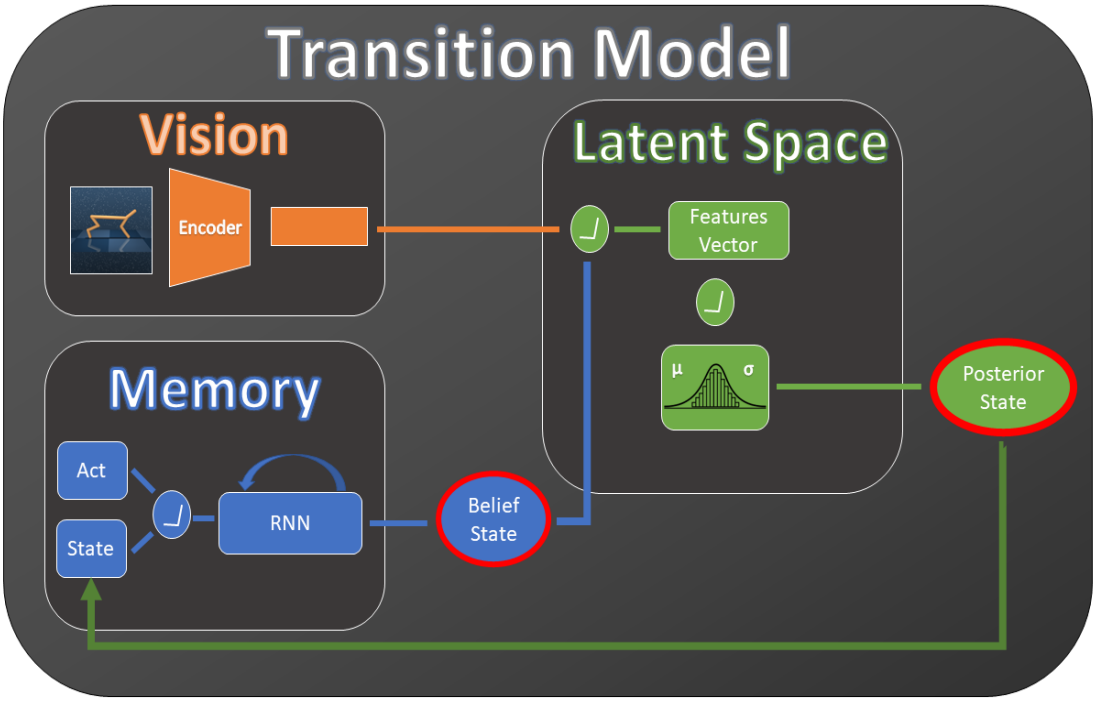



Figure 5.1: The transition model at inference time. The current frame is encoded and the RNN produces the current Belief State encoding the current action and the previous posterior state. The current encoded observation and the Belief State are combined to produce the Features Vector from where the posterior Gaussian parameters are produced. In the last step, the current Posterior state is sampled from the Gaussian.

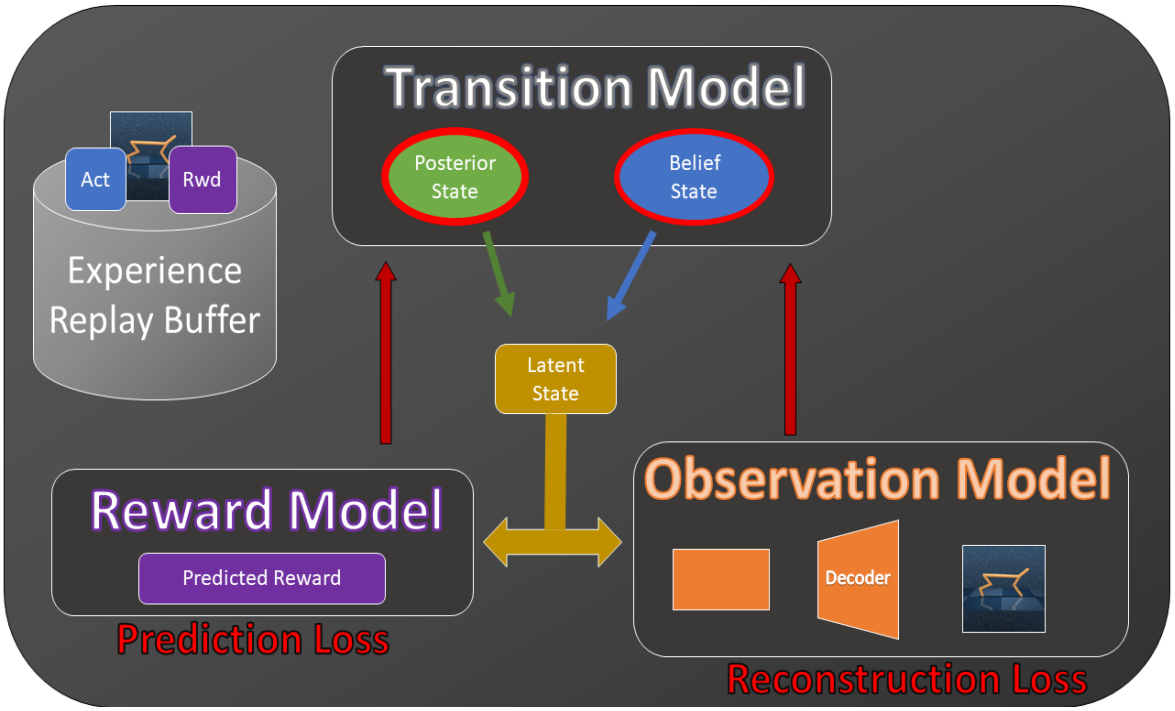



Figure 5.2: The current Latent State is produced by the combination of both Belief State and Posterior State. This Latent State is then used by the Reward Model to predict the reward and by the Observation Model to reconstruct the current observation. With these two results we can calculate the mean squared error (by sampling the original result from the buffer) and backpropagate the loss to train the transition model.

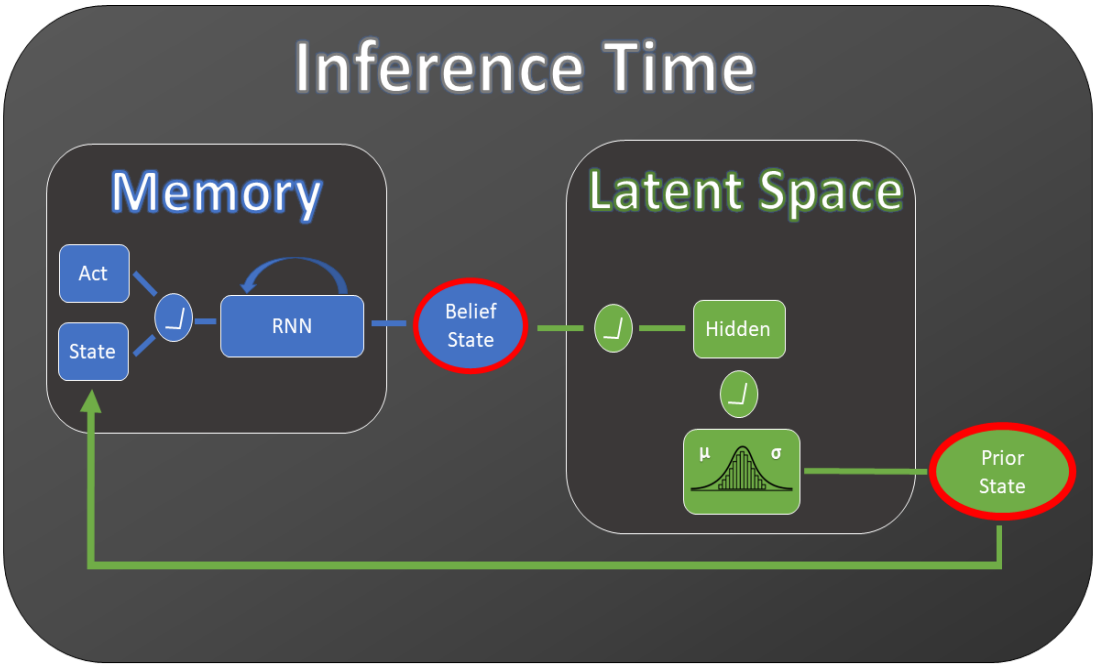



Figure 5.3: At inference time we have no more the experience replay buffer that provide us the observation for each step. We only have the observation for the current step provided by the environment and we have to predict the next for many steps ahead.

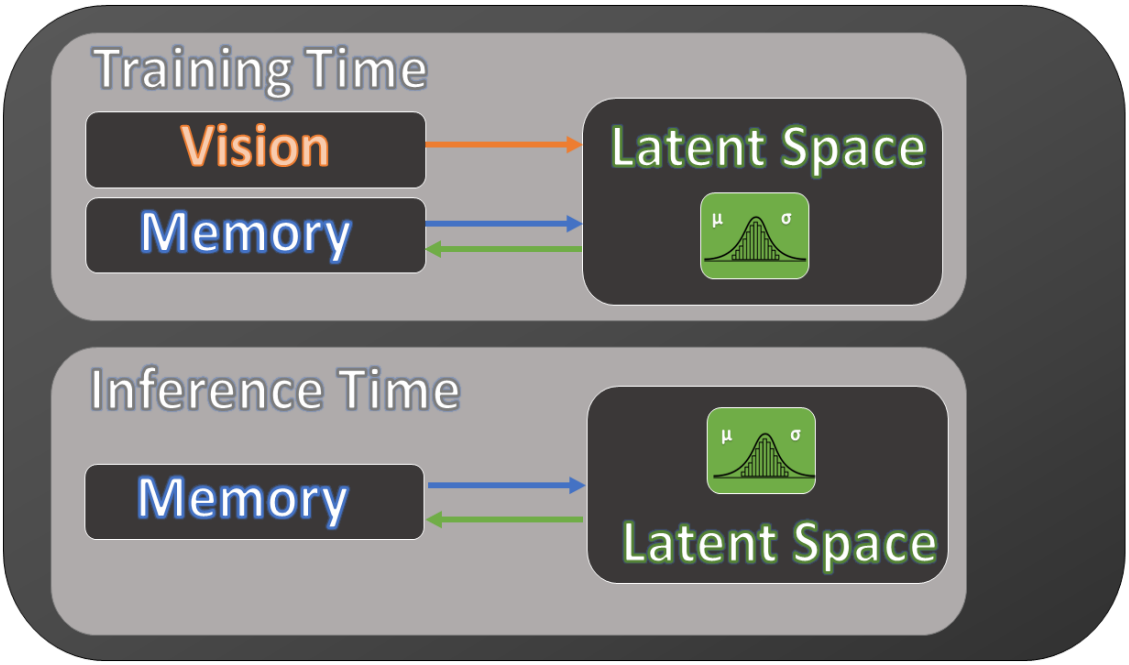



Figure 5.4: We can reuse the Memory model (RNN) used for the transition model at training time but we need to retrain the Gaussian model. We need a way to obtain the same parameters used by the model at training time.

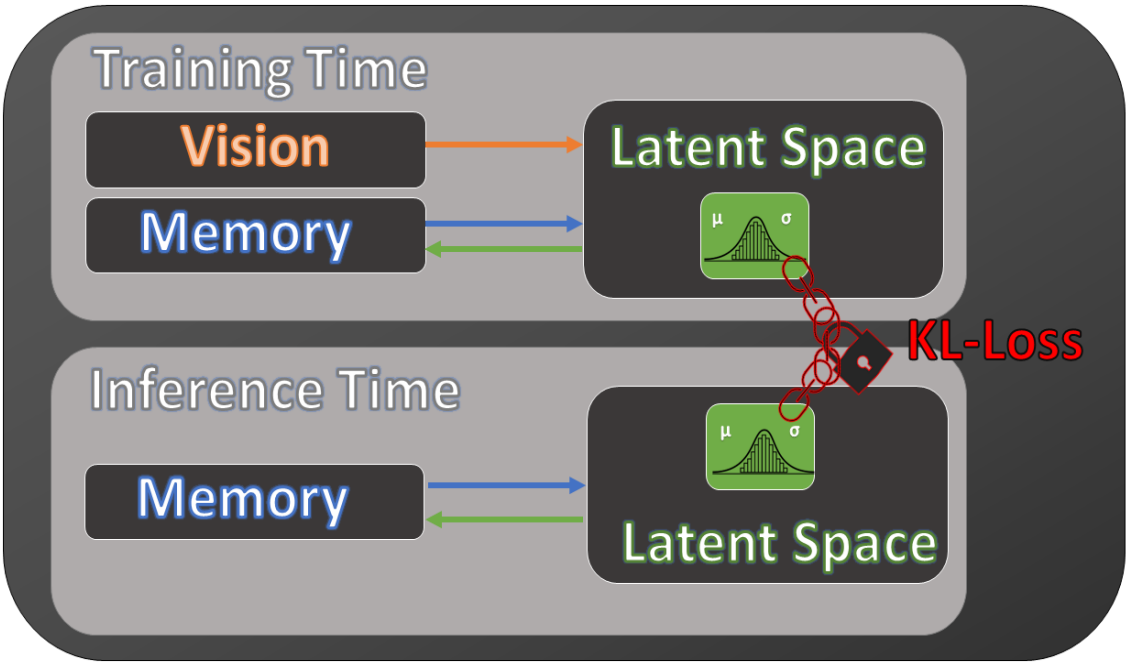



Figure 5.5: The loss indicates how much information we lost by approximate the Gaussian produced at training time with the one used at inference time.

### 5.2.2 Planning

Now it's time to use the model for planning. Even if the model that predicts the future is robust, a perfect prediction over the entire episode is unrealistic. The more we try to predict in the future, the more the prediction error will accumulate, and the more the prediction will diverge to reality. For this reason, the planning is computed over a short horizon $H$. They used the Cross-Entropy Method to perform trajectory optimization. It is a robust method and is proved to be capable of solving all the tested environments when true dynamics are given. Initially, the actions vector, that contains all the actions from the current time step $t$ to the planning horizon $H$, is sampled from a Gaussian with zero mean and unit variance $a_{t:t+H} \sim \text{Normal}\left(\mu_{t:t+H}, \sigma^2_{t:t+H}\right)$. For each generation, $J$ candidates action vectors are sampled and evaluated using the transition model and the reward model. The evaluation is base on how much reward is produced over the time steps. For each generation, the parameters update of the Gaussian is calculated over the top k elements of the candidates' population. Even if the planner has produced a plan over H time steps when the first action is executed, and the new observation is received, the planning process is replicated and adapted to the latest information. In other words, the planning is computed at every step, and only the first planned action is used. It is still necessary to planning over a horizon longer than one because that will lead to local optima.

## 5.3 Cross Entropy Method

The Cross-entropy (CE) method is an EDA (Estimation of Distribution Algorithms) used in many optimization problems of the form:

$$w^* = \arg\max_w S(w)$$

where w is a set o weight, and S is a generic objective function of w. The EDA is a specific family of Genetic Algorithms that does not work with a single solution but distributions of possible solutions represented with a covariance matrix $\Sigma$. This covariance matrix is used to defines a multivariate Gaussian function and for sampling the population for the next iteration. Iterations after iterations, the ellipsoid defined by $\Sigma$ is moved to the top part of the hill corresponding to the local optimum $\theta^*$. A each time step the entire population is sampled from the current parameters of the distributions. Next So all the new individuals are evaluated according to the problem-dependent fitness function $(f_i)_{i=1,\dots,\lambda}$. Then the top $K_e$ individuals $(z_i)_{i=1,\dots,K_e}$ (called **elites** or **candidates**) are used to update the distribution parameters (the new mean and variance are calculated over the elites).

$$\mu_{new} = \sum_{i=1}^{K_e} \lambda_i z_i$$
$$\Sigma_{new} = \sum_{i=1}^{K_e} \lambda_i \left(z_i - \mu_{old}\right)\left(z_i - \mu_{old}\right)^T + \epsilon\mathcal{I},$$

Notice that $(\lambda_i)_{i=1,\dots,K_e}$ are weights assigned to each individual (a common choice is $\lambda_i = \frac{1}{K_e}$. Usually some extra variance $\epsilon$ is added in order to prevent premature convergence. To be more specific some Gaussian noise is added to each individual $x_i$ that is sampled from the current covariance matrix $\Sigma$.

**Algorithm 4** Latent planning with CEM

| | | | |
|---|---|---|---|
| **Input:** | $H$ Planning horizon distance | $q(s_t\|o_{\leq t}, a_{<t})$ | Current state belief |
| | $I$ Optimization iterations | $p(s_t\|s_{t-1}, a_{t-1})$ | Transition model |
| | $J$ Candidates per iteration | $p(r_t\|s_t)$ | Reward model |
| | $K$ Number of top candidates to fit | | |

Initialize factorized belief over action sequences $q\left(a_{t:t+H}\right) \leftarrow \text{Normal}(0, \mathbb{I})$.
**for** optimization iteration $i = 1..I$
*// Evaluate J action sequences from the current belief.*
**for** candidate action sequence $j = 1..J$
$a^{(j)}_{t:t+H} \sim q(a_{t:t+H})$
$s^{(j)}_{t:t+H+1} \sim q(s_t|o_{1:t}, a_{1:t-1}) \prod_{\tau=t+1}^{t+H+1} p(s_\tau|s_{\tau-1}, a^{(j)}_{\tau-1})$
$R^{(j)} = \sum_{\tau=t+1}^{t+H+1} Ep(r_\tau|s^{(j)}_\tau)$
*//Re-fit belief to the K best action sequences.*
$\mathcal{K} \leftarrow \text{argsort}(\{R^{(j)}\}_{j=1}^{J})_{1:K}$
$\mu_{t:t+H} = \frac{1}{K}\sum_{k\in\mathcal{K}} a^{(k)}_{t:t+H}, \quad \sigma_{t:t+H} = \frac{1}{K-1}\sum_{k\in\mathcal{K}} |a^{(k)}_{t:t+H} - \mu_{t:t+H}|.$
$q(a_{t:t+H}) \leftarrow \text{Normal}(\mu_{t:t+H}, \sigma^2_{t:t+H}\mathbb{I})$
**return** first action mean $\mu_t$.

### 5.3.1 Algorithm

Finally, we have all the information to describe the entire flow of the Planet algorithm. Initially, some random episodes ( every action is chosen randomly) are executed in order to collect some data in the experience replay buffer. Then the main training loop, which is composed of two procedures called model fitting and data collection, can begin. **The model fitting procedure** consists of sampling sequence chunks from the buffer experience and train the model. **The data collection procedure** consists of using the model to solve an episode and collect new data. Since the aim of this procedure is not to solve the environment but collect new data, random Gaussian noise is added over the action before it is executed to have a better exploration of the environment. This noise is not used when we want to use/evaluate the model.

This iterative approach allows the model to collect also the data that is not obtainable from the random init episodes.

**Algorithm 5** Deep Planning Network (PlaNet)

**Input:**

| | | | |
|---|---|---|---|
| $R$ | Action repeat | $p(s_t \mid s_{t-1}, a_{t-1})$ | Transition model |
| $S$ | Seed episodes | $p(o_t \mid s_t)$ | Observation model |
| $C$ | Collect interval | $p(r_t \mid s_t)$ | Reward model |
| $B$ | Batch size | $q(s_t \mid o_{\leq t}, a_{<t})$ | Encoder |
| $L$ | Chunk length | $p(\epsilon)$ | Exploration noise |
| $\alpha$ | Learning rate | | |

Initialize dataset $\mathcal{D}$ with $S$ random seed episodes.
Initialize model parameters $\theta$ randomly.
**while**(not converged)

// Model fitting

**for** update step $s = 1..C$

Draw sequence chunks $\{(o_t, a_t, r_t)_{t=k}^{L+k}\}_{i=1}^{B} \sim \mathcal{D}$ uniformly at random from the dataset.

Compute loss $\mathcal{L}(\theta)$ .

Update model parameters $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta)$.

// Data Collection

$o_1 \leftarrow$ `env.reset()`

**for** time step $t = 1.. \lceil \frac{T}{R} \rceil$ **do**

Infer belief over current state $q(s_t \mid o_{\leq t}, a_{<t})$ from the history.

$a_t \leftarrow$ `planner`$(q(s_t \mid o_{\leq t}, a_{<t}), p)$ see 4 for details

Add exploration noise $\epsilon \sim p(\epsilon)$ to the action.

**for** action repeat $k = 1..R$

$r_t^k, o_{t+1}^k \leftarrow$ `env.step`$(a_t)$

$r_t, o_{t+1} \leftarrow \sum_{k=1}^{R} r_t^k, o_{t+1}^R$

$\mathcal{D} \leftarrow \mathcal{D} \cup \{(o_t, a_t, r_t)_{t=1}^{T}\}$

# Chapter 6

# Experiments

In this chapter, we introduce the benchmark environments used to compute all the tests. We show the results obtained, the proposal for improvement, and a comparison between the model-free approach represented by the DDPG algorithm and the model-based approach represented by PlaNet.

## 6.1 DeepMind Control Suite

All the experiments are based on Deepmind Control Suite [8]. The control suite is a set of continuous control tasks that are built for benchmarking reinforcement learning agents. The main focus is on continuous control. The environments can provide a fully observable state (a feature vectors) and a partially observable state (a scene) frame. All the environments are written in Python and powered by the MuJoCo physics engine. A visual representation of the main environments available in the Deepmind control suite is shown below.

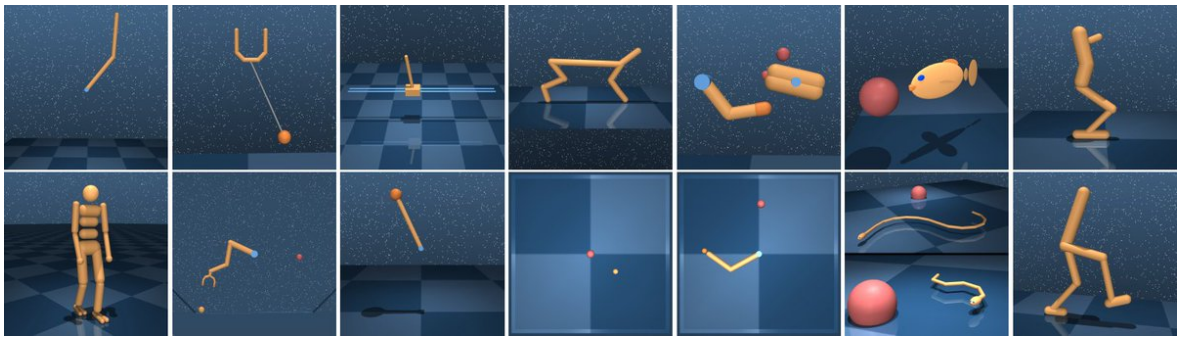


Figure 6.1: . Top: Acrobot, Ball-in-cup, Cart-pole, Cheetah, Finger, Fish, Hopper. Bottom: Humanoid, Manipulator, Pendulum, Point-mass, Reacher, Swimmer (6 and 15 links), Walker.

The principal environment that we choose to experiment is the half-cheetah that is a very common choice.

In this environment, the agent should move forward as quickly as possible with a cheetah like body that is constrained to the plane. The reward is linearly pro-

portional to a maximum of $10_{\mathrm{m/s}}$ i.e. $r(v) = \max(0, \min(v/10, 1))$. A vector of 18 dimensions describes each state while the actions are represented with a vector of 6 dimensions.

In addition to this environment, we have chosen three more to test the consistency of the tests.

The other environments are:

- Cart-pole (task: swing-up): The classic cart-pole swing-up task. The agent must balance a pole attached to a cart by applying forces to the cart alone. The pole starts each episode hanging upside-down.
- Walker (task: walk): Agent should move forward as quickly as possible with a bipedal walker constrained to the plane without falling or pitching the torso too far forward or backward.
- Reacher: (mode: easy): The simple two-link planar reacher with a randomized target location. The reward is one when the end effector penetrates the target sphere.

## 6.2 Model Free experiments

We choose DDPG as a model-free algorithm for the experiments since it is compatible with environments with continuous action spaces. It also guarantees a good sample efficiency thanks to the buffer experience replay.

In this experiment we want to find out what level of performances a DDPG agent can reach with a million of steps. We start from the original DDPG paper [10] but when it was released, the Deepmind control suite did not exist yet. All the benchmarks in that paper, for the cheetah problem, are based on another suite provided by Open Ai called Gym. Even if both the environments from Open Ai and Deepmind are based on the same physic engine (MuJoCo) and representing the same problem (the cheetah problem), they have significant differences that require a different set of parameters. All the parameters provided in the original paper are based on the Open Ai Gym version.

We addressed this problem in two phases. At first, we precisely replicated the original paper model with the Open Ai Gym environment to be sure to have a solid implementation. We tried to retrain our model in the Deepmind Control Suite environment without the tuning process, without success. The DDPG algorithm proved to be very susceptible to the parameters. We tried another approach based on the Deepmind Control Suite paper in which the author explained how they trained the DDPG algorithm in their environments, and we are successfully reproduced their results.

In our implementation, we used Adam ([24]) for learning the neural network parameters with a learning rate of $10^{-4}$ for both the actor and critic networks. For the critic network, we included a $L_2$ weight decay of 0.002 and used a discount factor of $\gamma = 0.99$. We used both a soft update (with $\tau = 0.001$) and a hard update (every 100 steps) for the target networks. The activation function is the Relu for all the

hidden layers and the Tanh for the actor final output layer. After the activation, I apply batch normalization. In the final layer (for both actor and critic networks) both the weight and bias are initialized from uniform distribution $\left[-3 \times 10^{-3}, 3 \times 10^{-3}\right]$. The hidden layers instead are initialized from uniform distributions $\left[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}\right]$ where $f$ is the fan-in of the layer. The actor-network is composed of 3 hidden layers with respectively [128,300,400] units. Only for the actor-network, the gradients are clipped at [-1,1]. The critic-network is composed of two separate input layers (one for the action with 256 units and two for the state with 128,256 units). Then the two activations are summed together and passed to another 2 hidden layers with 300 and 400 units. Lastly we used an Ornstein-Uhlenbeck process to produce the noise for the exploration. The parameters are: $\theta = 0.15$ and $\sigma = 0.3$. We do not perform warm-up episodes to prefill the buffer before training.

The results of our experiments are shown below.

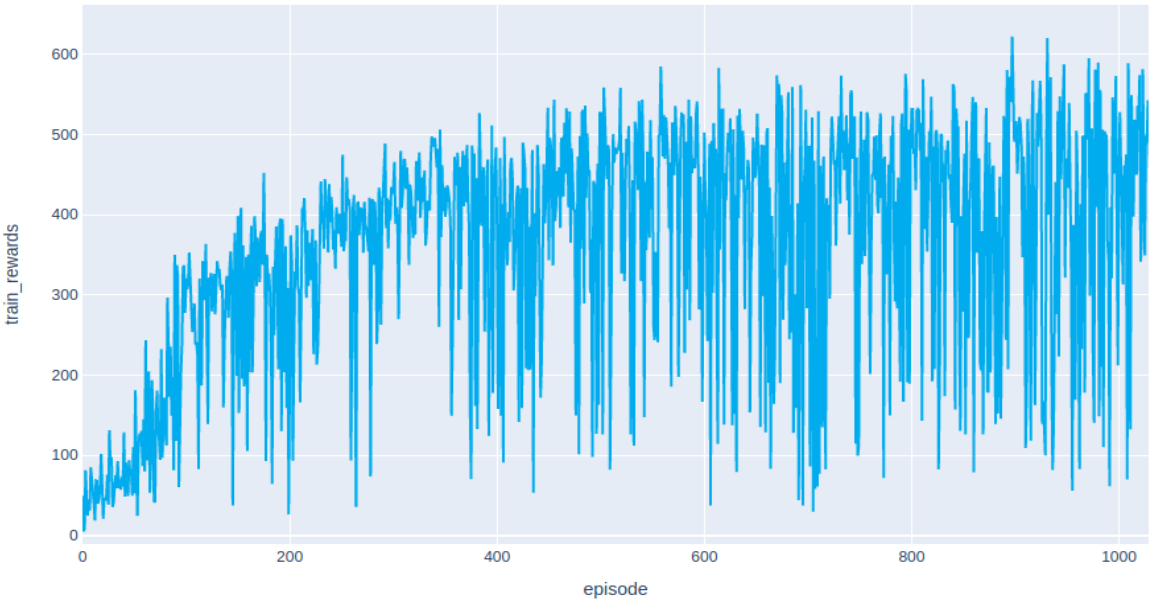



Figure 6.2: Results of the training of the DDPG algorithm on DeepMind Control Suite Ceetah environment.

Every episode corresponds to 1000 steps, and the training consists of 1 million steps. During the training, at every action is added a gaussian noise. To understand the real performance of the model, a test without noise is computed every 100 episodes.

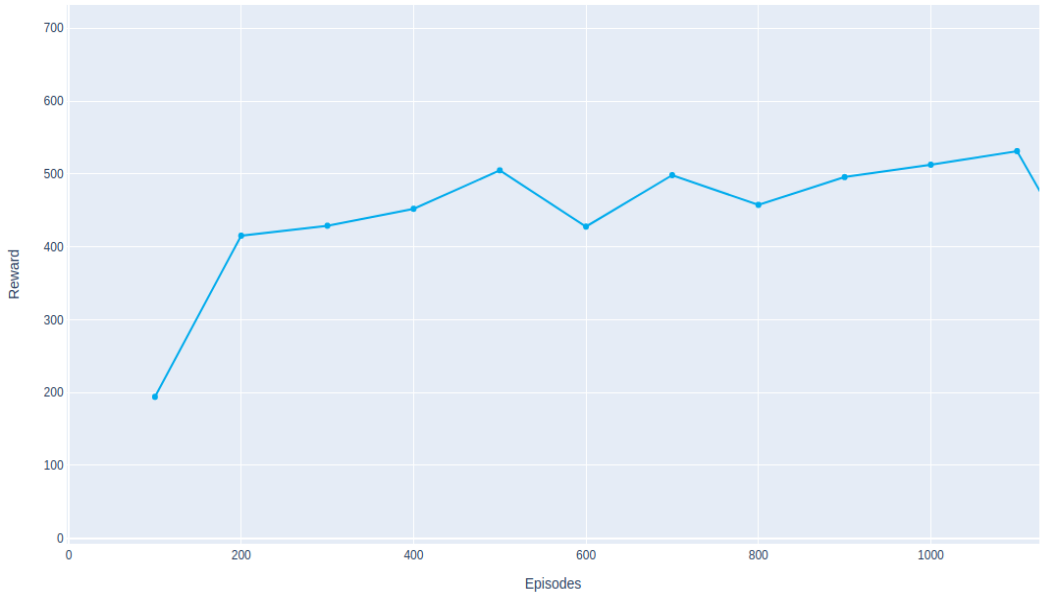



Figure 6.3: Results of the test with the model trained with the DDPG algorithm from feature vectors.

This result is consistent with the performance published by the Deepmind Control Suite authors [8].

| Domain | Task | A3C | D4PG | D4PG (Pixels) | DDPG |
|---|---|---|---|---|---|
| acrobot | swingup | 41.9 ± 1.2 | 297.6 ± 8.4 | 81.7 ± 4.4 | 15.4 ± 0.9 |
| | swingup_sparse | 0.2 ± 0.1 | 198.7 ± 9.1 | 13.0 ± 1.7 | 1.0 ± 0.1 |
| ball_in_cup | catch | 104.7 ± 7.8 | 981.2 ± 0.7 | 980.5 ± 0.5 | 984.5 ± 0.3 |
| cartpole | balance | 951.6 ± 2.4 | 966.9 ± 1.9 | 992.8 ± 0.3 | 917.4 ± 2.2 |
| | balance_sparse | 857.4 ± 7.9 | 1000.0 ± 0.0 | 1000.0 ± 0.0 | 878.5 ± 8.0 |
| | swingup | 558.4 ± 6.8 | 845.5 ± 1.2 | 862.0 ± 1.1 | 521.7 ± 6.1 |
| | swingup_sparse | 179.8 ± 5.9 | 808.4 ± 4.4 | 482.0 ± 56.6 | 4.5 ± 1.1 |
| cheetah | walk | 213.9 ± 1.6 | 736.7 ± 4.4 | 523.8 ± 6.8 | 842.5 ± 1.6 |
| finger | spin | 129.4 ± 1.5 | 978.2 ± 1.5 | 985.7 ± 0.6 | 920.3 ± 6.3 |
| | turn_easy | 167.3 ± 9.6 | 983.4 ± 0.6 | 971.4 ± 3.5 | 942.9 ± 4.3 |
| | turn_hard | 88.7 ± 7.3 | 974.4 ± 2.8 | 966.0 ± 3.4 | 939.4 ± 4.1 |
| fish | swim | 81.3 ± 1.1 | 844.3 ± 3.1 | 72.2 ± 3.0 | 492.7 ± 9.8 |
| | upright | 474.6 ± 6.6 | 967.4 ± 1.0 | 405.7 ± 19.6 | 854.8 ± 3.3 |
| hopper | hop | 0.5 ± 0.0 | 560.4 ± 18.2 | 242.0 ± 2.1 | 501.4 ± 6.0 |
| | stand | 27.9 ± 2.3 | 954.4 ± 2.7 | 929.9 ± 3.8 | 857.7 ± 2.2 |
| humanoid | run | 1.0 ± 0.0 | 463.6 ± 13.8 | 1.4 ± 0.0 | 167.9 ± 4.1 |
| | stand | 6.0 ± 0.1 | 946.5 ± 1.8 | 8.6 ± 0.2 | 642.6 ± 2.1 |
| | walk | 1.6 ± 0.0 | 931.4 ± 2.3 | 2.6 ± 0.1 | 654.2 ± 3.9 |
| manipulator | bring_ball | 0.4 ± 0.0 | 895.9 ± 3.7 | 0.5 ± 0.1 | 0.6 ± 0.1 |
| pendulum | swingup | 48.6 ± 5.2 | 836.2 ± 5.0 | 680.9 ± 41.9 | 816.2 ± 4.7 |
| point_mass | easy | 545.3 ± 9.3 | 977.3 ± 0.6 | 977.8 ± 0.5 | 618.0 ± 11.4 |
| reacher | easy | 95.6 ± 3.5 | 987.1 ± 0.3 | 967.4 ± 4.1 | 917.9 ± 6.2 |
| | hard | 39.7 ± 2.9 | 973.0 ± 2.0 | 957.1 ± 5.4 | 904.3 ± 6.8 |
| swimmer | swimmer15 | 164.0 ± 7.3 | 658.4 ± 10.0 | 180.8 ± 11.9 | 421.8 ± 13.5 |
| | swimmer6 | 177.8 ± 7.8 | 664.7 ± 11.1 | 194.7 ± 15.9 | 394.0 ± 14.1 |
| walker | run | 191.8 ± 1.9 | 839.7 ± 0.7 | 567.2 ± 18.9 | 786.2 ± 0.4 |
| | stand | 378.4 ± 3.5 | 993.1 ± 0.3 | 985.2 ± 0.4 | 969.8 ± 0.3 |
| | walk | 311.0 ± 2.3 | 982.7 ± 0.3 | 968.3 ± 1.8 | 976.3 ± 0.3 |

Table 1: Mean and Standard Error of 100 episodes after $10^8$ training steps for each seed.

Figure 6.4: Result of the experiments from the Deepmind Control Suite team [8].

As we can see from the image above our implementation outperform the A3C algorithm and reach a cumulative reward value of 500 after $10^6$ steps that is compatible with the 812.5 obtained from the research team after $10^8$ steps.

After our experiments, we can say that the DDPG algorithm is proved to be very sensitive to the hyperparameters. We notice that some parameters like the initialization of the layers, the learning rate are more impact respect to the others.

## 6.3 Model Free experiments from frames

The use of the features vectors requires a human expert's intervention. This can be a limitation and also a source of error in the construction of the environment.

In this experiment, we want to find out if the DDPG algorithm is capable of solving this task directly from the raw pixels and in that case, how much the difficult of the problem increase.

With this new formulation, the observation provided does not correspond to the real markovian state of the MDP. The authors of the Deepmind Control Suite doesn't use the DDPG algorithm to solve this problem but switched to an advanced version of the algorithm called Distributed Distributional Deterministic Policy Gradients (**D4PG**). They showed that this version of the algorithm after $10^8$ steps, is able to learn a policy also in this condition, but is not capable of achieving the same performances of the experiments with features vector as input.

| Domain | Task | A3C | D4PG | D4PG (Pixels) | DDPG |
|---|---|---|---|---|---|
| acrobot | swingup | 41.9 ± 1.2 | 297.6 ± 8.4 | 81.7 ± 4.4 | 15.4 ± 0.9 |
| | swingup_sparse | 0.2 ± 0.1 | 198.7 ± 9.1 | 13.0 ± 1.7 | 1.0 ± 0.1 |
| ball_in_cup | catch | 104.7 ± 7.8 | 981.2 ± 0.7 | 980.5 ± 0.5 | 984.5 ± 0.3 |
| cartpole | balance | 951.6 ± 2.4 | 966.9 ± 1.9 | 992.8 ± 0.3 | 917.4 ± 2.2 |
| | balance_sparse | 857.4 ± 7.9 | 1000.0 ± 0.0 | 1000.0 ± 0.0 | 878.5 ± 8.0 |
| | swingup | 558.4 ± 6.8 | 845.5 ± 1.2 | 862.0 ± 1.1 | 521.7 ± 6.1 |
| | swingup_sparse | 179.8 ± 5.9 | 808.4 ± 4.4 | 482.0 ± 56.6 | 4.5 ± 1.1 |
| cheetah | walk | 213.9 ± 1.6 | 736.7 ± 4.4 | 523.8 ± 6.8 | 842.5 ± 1.6 |
| finger | spin | 129.4 ± 1.5 | 978.2 ± 1.5 | 985.7 ± 0.6 | 920.3 ± 6.3 |
| | turn_easy | 167.3 ± 9.6 | 983.4 ± 0.6 | 971.4 ± 3.5 | 942.9 ± 4.3 |
| | turn_hard | 88.7 ± 7.3 | 974.4 ± 2.8 | 966.0 ± 3.4 | 939.4 ± 4.1 |
| fish | swim | 81.3 ± 1.1 | 844.3 ± 3.1 | 72.2 ± 3.0 | 492.7 ± 9.8 |
| | upright | 474.6 ± 6.6 | 967.4 ± 1.0 | 405.7 ± 19.6 | 854.8 ± 3.3 |
| hopper | hop | 0.5 ± 0.0 | 560.4 ± 18.2 | 242.0 ± 2.1 | 501.4 ± 6.0 |
| | stand | 27.9 ± 2.3 | 954.4 ± 2.7 | 929.9 ± 3.8 | 857.7 ± 2.2 |
| humanoid | run | 1.0 ± 0.0 | 463.6 ± 13.8 | 1.4 ± 0.0 | 167.9 ± 4.1 |
| | stand | 6.0 ± 0.1 | 946.5 ± 1.8 | 8.6 ± 0.2 | 642.6 ± 2.1 |
| | walk | 1.6 ± 0.0 | 931.4 ± 2.3 | 2.6 ± 0.1 | 654.2 ± 3.9 |
| manipulator | bring_ball | 0.4 ± 0.0 | 895.9 ± 3.7 | 0.5 ± 0.1 | 0.6 ± 0.1 |
| pendulum | swingup | 48.6 ± 5.2 | 836.2 ± 5.0 | 680.9 ± 41.9 | 816.2 ± 4.7 |
| point_mass | easy | 545.3 ± 9.3 | 977.3 ± 0.6 | 977.8 ± 0.5 | 618.0 ± 11.4 |
| reacher | easy | 95.6 ± 3.5 | 987.1 ± 0.3 | 967.4 ± 4.1 | 917.9 ± 6.2 |
| | hard | 39.7 ± 2.9 | 973.0 ± 2.0 | 957.1 ± 5.4 | 904.3 ± 6.8 |
| swimmer | swimmer15 | 164.0 ± 7.3 | 658.4 ± 10.0 | 180.8 ± 11.9 | 421.8 ± 13.5 |
| | swimmer6 | 177.8 ± 7.8 | 664.7 ± 11.1 | 194.7 ± 15.9 | 394.0 ± 14.1 |
| walker | run | 191.8 ± 1.9 | 839.7 ± 0.7 | 567.2 ± 18.9 | 786.2 ± 0.4 |
| | stand | 378.4 ± 3.5 | 993.1 ± 0.3 | 985.2 ± 0.4 | 969.8 ± 0.3 |
| | walk | 311.0 ± 2.3 | 982.7 ± 0.3 | 968.3 ± 1.8 | 976.3 ± 0.3 |

Table 1: Mean and Standard Error of 100 episodes after $10^8$ training steps for each seed.

Figure 6.5: Result of the Deepmind Control Suite team obtained with the D4PG algorithm[8].

We still tried to train a DDPG agent from raw pixels. As suggest in the original paper [10] we used the action repeats trick to enrich the information provided at each step. So at each step, the agent computes the same action 3 times. We have transformed the obtained 3 frames to grayscale, and we stacked them together the creating a new single input of 3 feature maps. The original version does not apply the grayscale conversion and provide an input of 9 feature maps. We notice anyway that this preprocessing step is very useful. All the frames are downsampled to 64x64 pixels and normalized in a range between [0,1]. We added a new set of convolutional layers to the model to handle the frames high dimension. We experimented with different approaches to network architecture. Initially, I tried to create two convolutional networks, one for the actor and another for the critic but did not work well. The best performance is obtained by weight-sharing of the convolutional network, as shown in the image below.

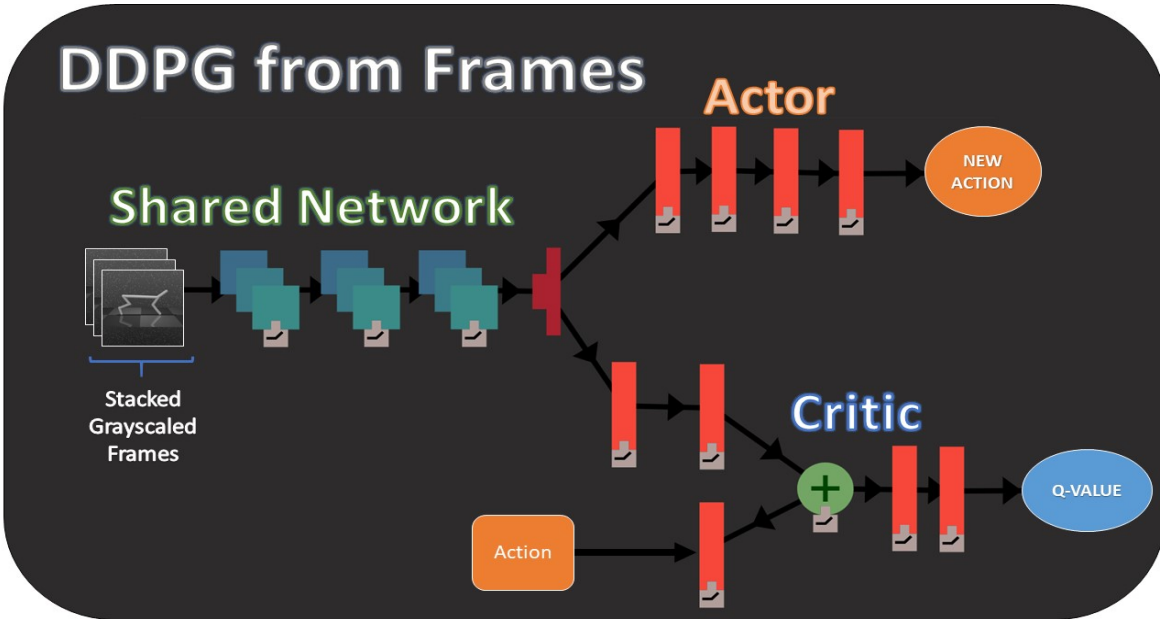



Figure 6.6: Visual representation of the model architecture with the convolutional network shared between actor and critic.

Whenever the actor or the critic receives a frame as input, they call the shared convolutional network to encode the frames and return the corresponding features vector. As suggested in [8], only the Critic network is allowed to update the shared network weights; in other words, the Actor gradients are truncated. The shared network is composed of three layers, all with a kernel size of 3 x 3 with 32 channels (only the first layer with has also a stride of 2), followed by two fully-connected layers with 200 and 50 neurons, with layer normalization. All other parameters are the same as in the previous experiment, except the batch size is set to 256.

We stop the training after $10^6$ steps like all the other experiments. The results are shown below.

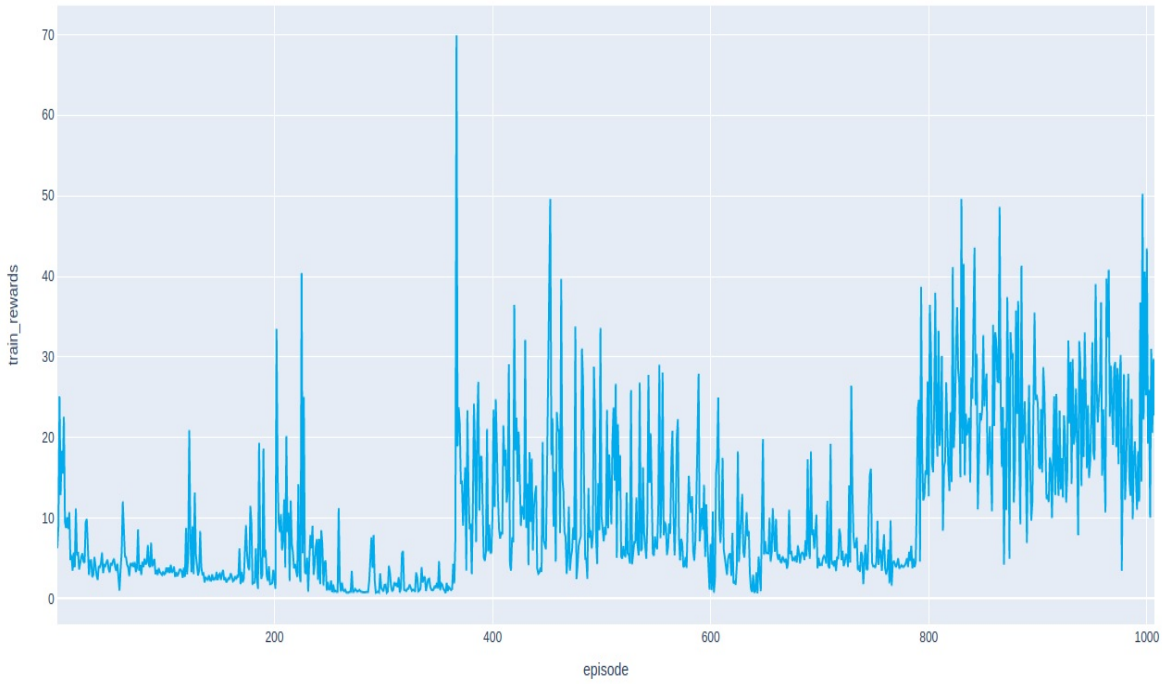



Figure 6.7: Result of training DDPG for the cheetah problem, after 1000 episodes using frames as input .

As we can see, after 1000 episodes (1 million steps), the agent is not able to reach significative performance. The training curve has risen slightly, indicating that full training would require several thousand more episodes. Due to the limits of the computational budget is was not possible to train the network entirely.

We suppose that the convolutional layers require a lot of transition before learning the useful pieces of information to capture from every image. Until the convolutional layers are not trained, the policy can't learn. We can clearly see how to learn from raw pixels is more complicated than learn from the features vector.

## 6.4 Model Based experiments

The algorithm chosen for the model-based experiments is Planet. We have not implemented it from scratch, but we build upon an open-source version available on GitHub.

We test the PlaNet algorithm on the same benchmark environment to see if this algorithm is able to solve in a million on steps the cheetah problem with raw pixels as input.

We do not operate tuning, so the parameters are the same as the open-source implementation, and we do not repeat them here.

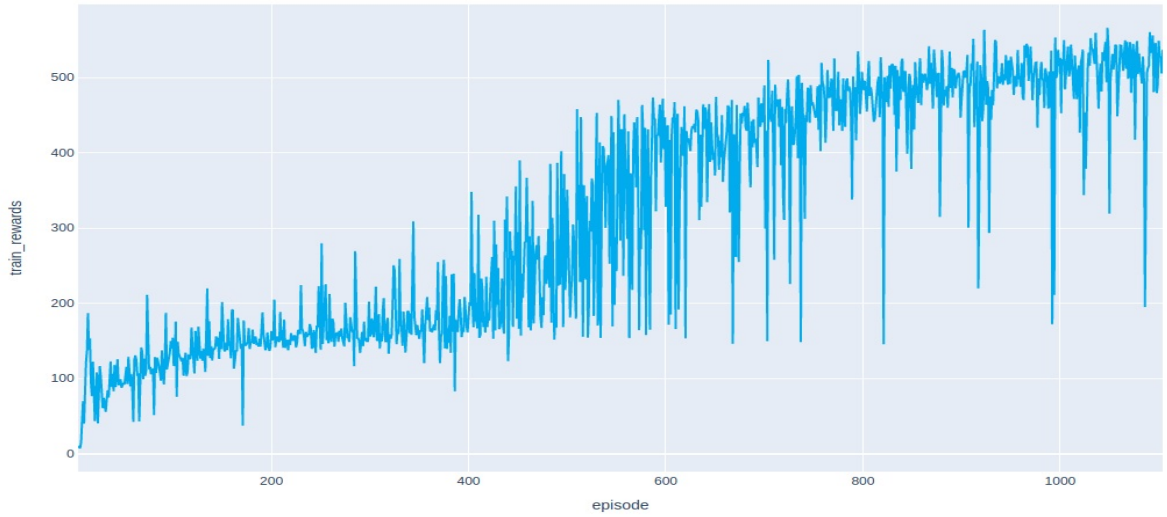



Figure 6.8: Performance obtained with the open source version of PlaNet at training time.

Like with the DDPG training, for every action, Gaussian noise is added, for this reason, there is a variance in the performance, but the training curve is monotonically increasing.

To find the real performance of the agent, we can see the test curve in which the same model is used without Gaussian noise.

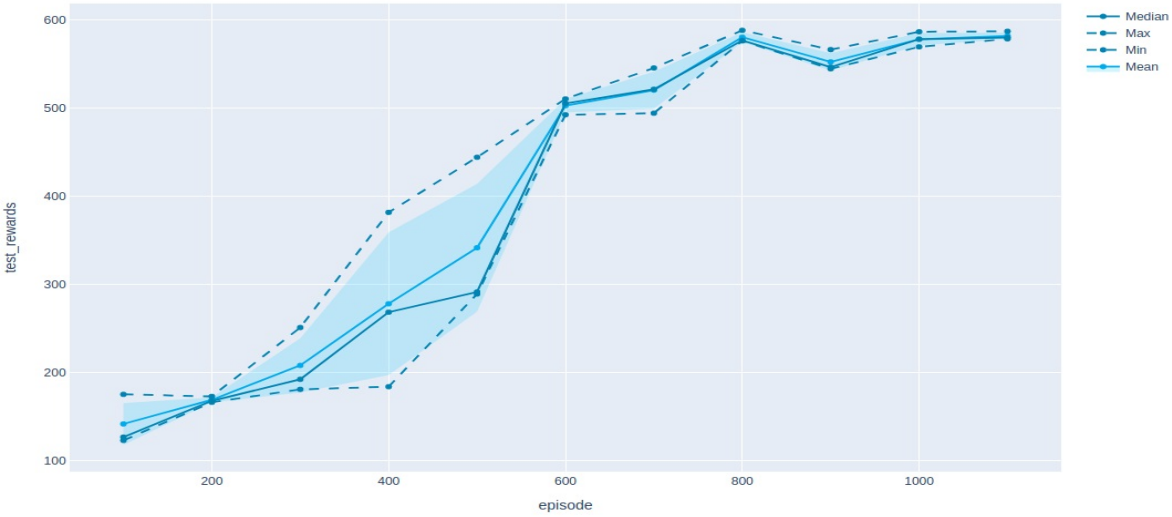



Figure 6.9: Performance obtained with the open source version of PlaNet at test time.

As we can see, after 1 million steps, the Planet agent is able to achieve a result of 578 rewards.

Beyond the final performances, it is interesting to deeply analyze the model predictions and the ability to generate predicted frames. So in this experiment, we focus on the visual component of the prediction model.

We only provided the first frame to the model, and it predicted all the rest without receiving any further information beyond the actions performed at each step.

Both the observations and the predicted frames are resized to 64 x 64 pixels in order to reduce the computational cost.

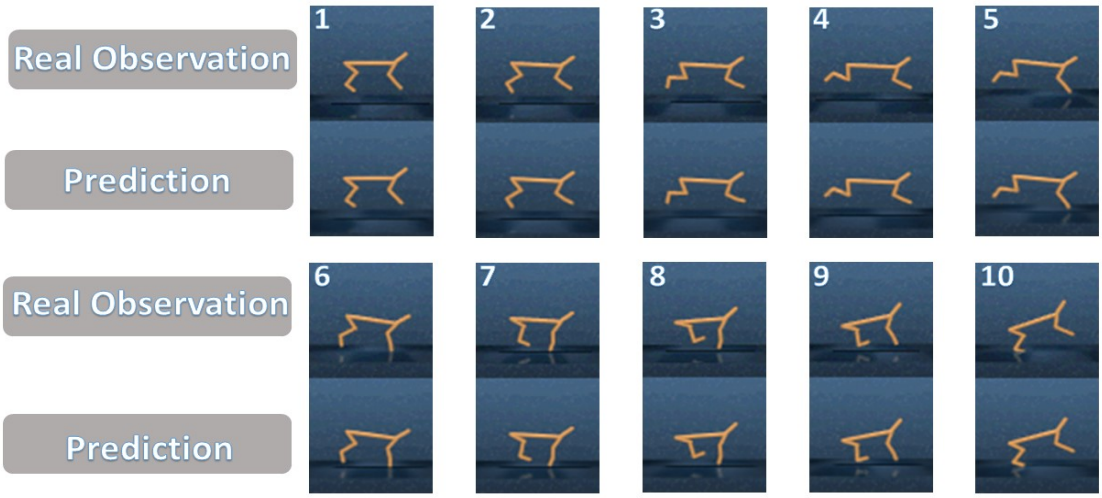



Figure 6.10: Comparison between the first 10 real observations (the top frame) and the 10 predictions (the bottom frame).

As we can see from the image above, the model is able to predict all the first 10 steps correctly, but it is hard to keep the memory of the past experience for a long time, so as the predictions go on, the errors pile up. Initially, these errors are barely perceptible. For this reason, we have extended the planning horizon up to 20 steps reaching the point where these errors are easily visible.

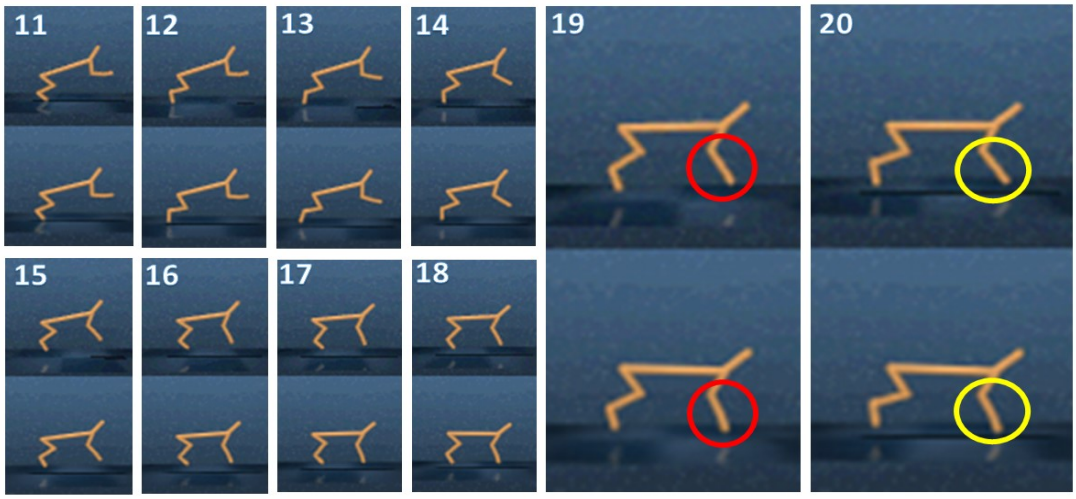



Figure 6.11: We can start to see discrepancies as the predictions goes on.

In order to make this comparison more clearly, we calculate the mean squared error (MSE) of each frames and we plot the pixels difference over the 20 planned steps.

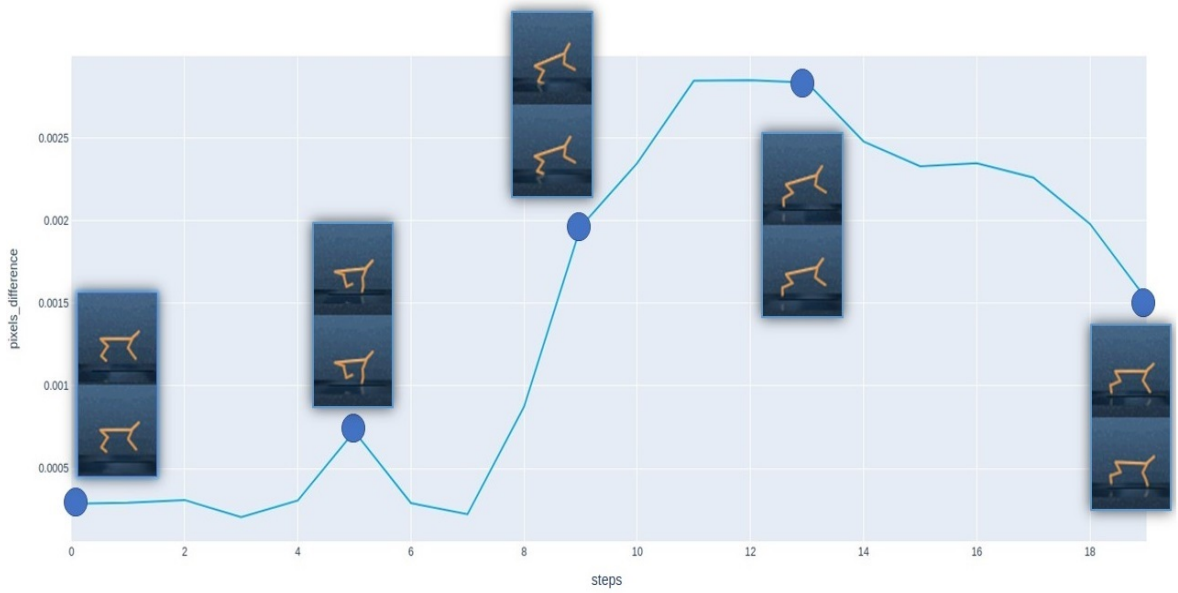



Figure 6.12: The mean squared error of the real and predicted frames over 20 steps.

This is not a problem for the planning algorithm since the model predict only for a short horizon. In particular the authors of PlaNet has chosen a planning horizon of 12 steps.

We have also produced a heatmap to highlight the area in which the model produce the most errors.

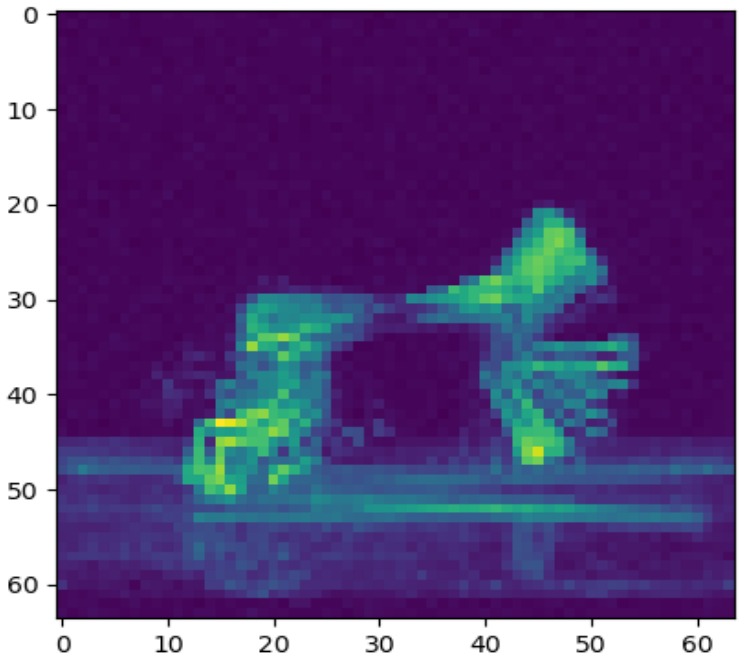



Figure 6.13: The heatmap highlight the area of all the predicted frames where the model has made the greatest errors.

As we could expect the heatmap show that the model make more error in the area of the in the hind and front legs of the cheetah and in the zone of the head.

We also tested the reward predictions. The plot below show how close are the predictions of the fully trained model and the effective received reward for an entire episode.


Figure 6.14: Comparison between the reward model predictions and the effective reward obtained.

## 6.5 Experiments with PlaNet

In this section we investigate the chance to improve the performance of the open-source version of PlaNet. We tried three different ways.

The first try is about the preprocessing phase of each frame steps. At each steps, the generated frame is preprocessed before being used as input to the model. In particular, operation of resizing is applied by the cv2 library using the INTER LINEAR algorithm. Exploring the DeepMind Control Suite code, we saw that this process could be avoided indicating directly to the camera the size of the frame to be rendered with the command: `self._env.physics.render(height=64, width=64, camera_id=0)`. We found that also the original implementation uses a resize method instead of native render in low dimensions. In particular they use the *"skimage.transform.resize"* method as you can see in their implementation.

We can see how changing this single one line of code has a huge positive impact on the final performance.

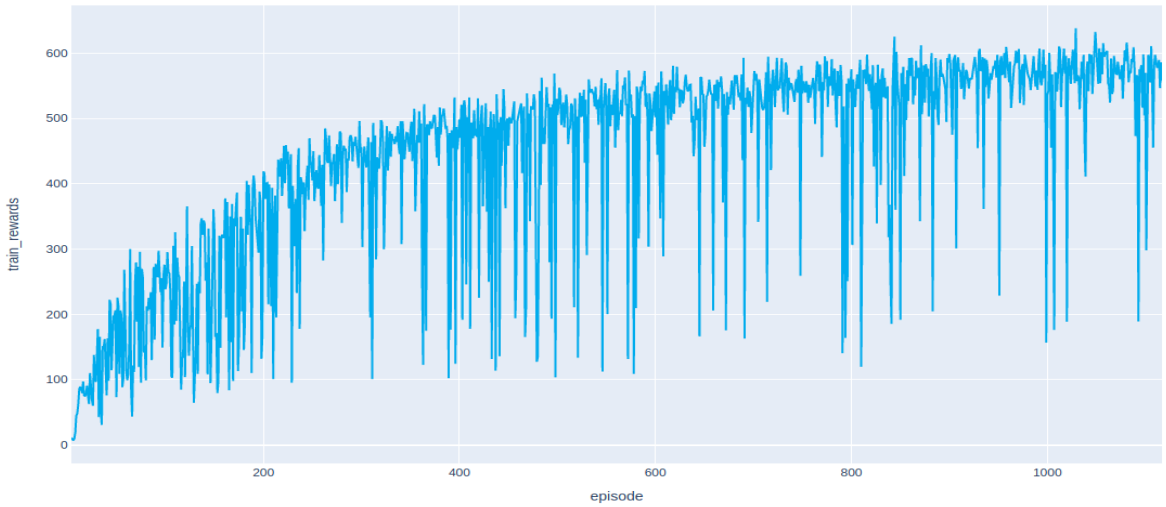



Figure 6.15: Performance obtained at training time without resize the frames.

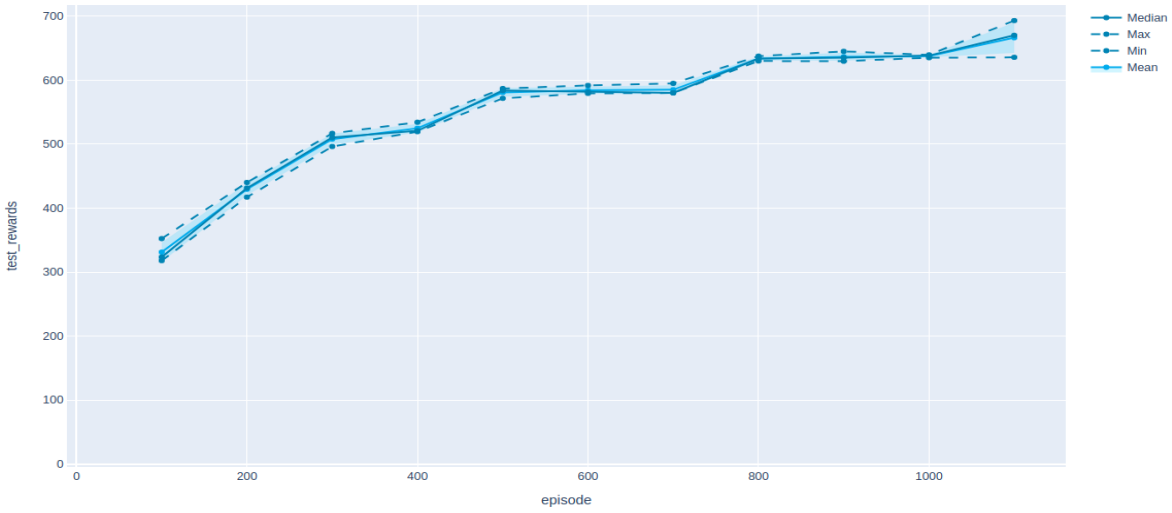



Figure 6.16: Performance obtained at test time without resize the frame.

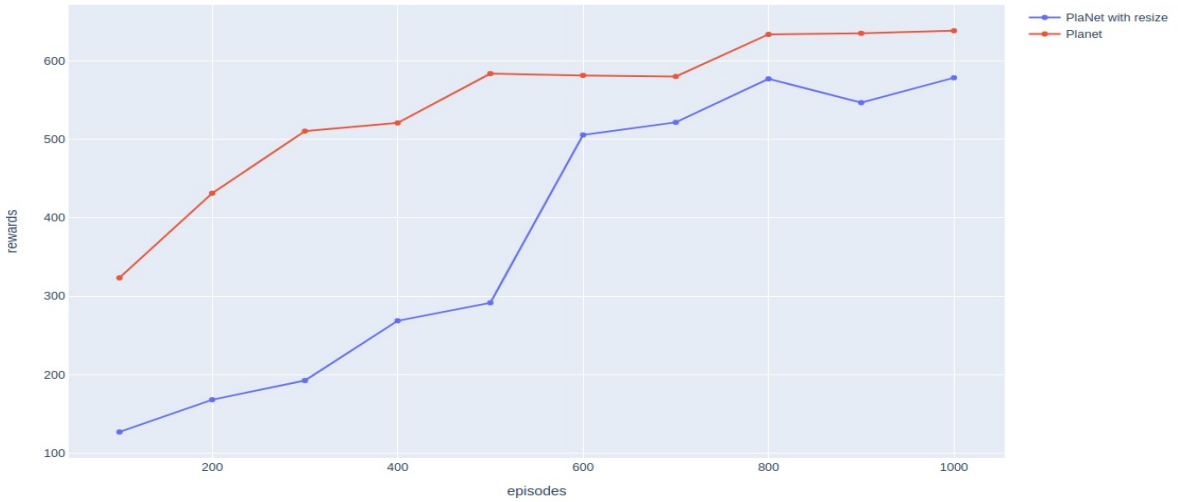



Figure 6.17: Comparison between the performance of PlaNet with and without the frames resizing.

We think that this improvement is due to the fact that some information is lost when the algorithm of resizing are applied in the preprocessing phase.
A second experiment is based on the idea of enriching the information at each step with the obtained reward, in addition to the current frame. During the experiments, we notice that when the model is not fully trained can happen that it keeps doing the same action believing to collect rewards. Indeed what really happening is that the agent keeps predicting a reward when in the real environment, it doesn't receive any good feedback. After some training iterations, the agent fits better the environment dynamics, and the problem disappears. So the idea is to explicitly provide also the received reward so the agent can use it to recognize, at inference time, that it's predictions are not consistent with what is really happening. So, we modify the model by concatenating the encoded current observation with the previous reward.

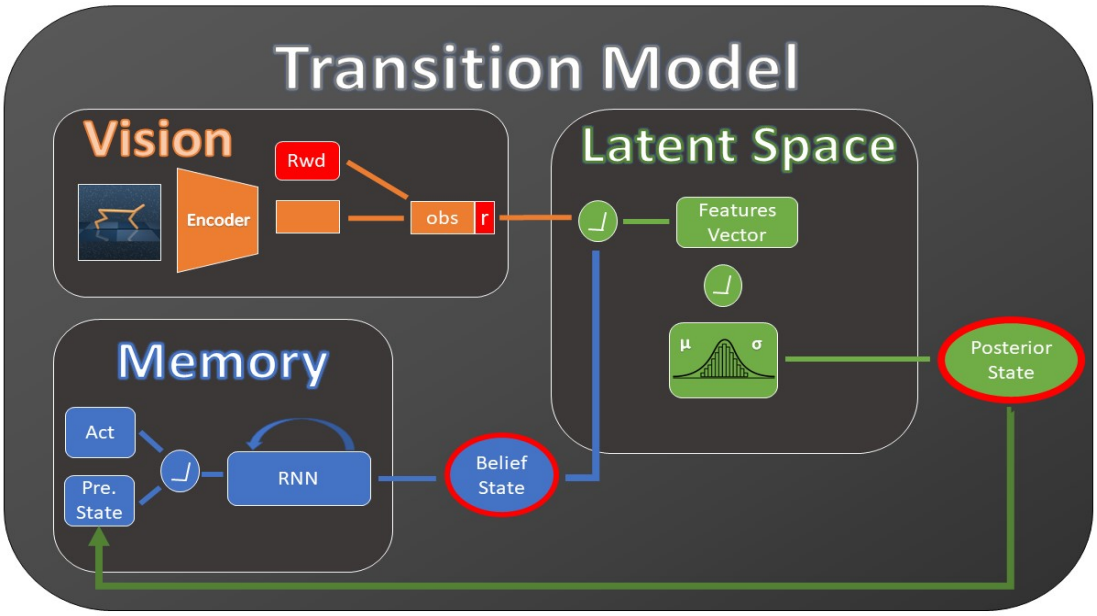



Figure 6.18: Concatenating the reward to the current observation.

The result of training seems to be not so promising. The training curve has more variance and does not overcome the previous version.

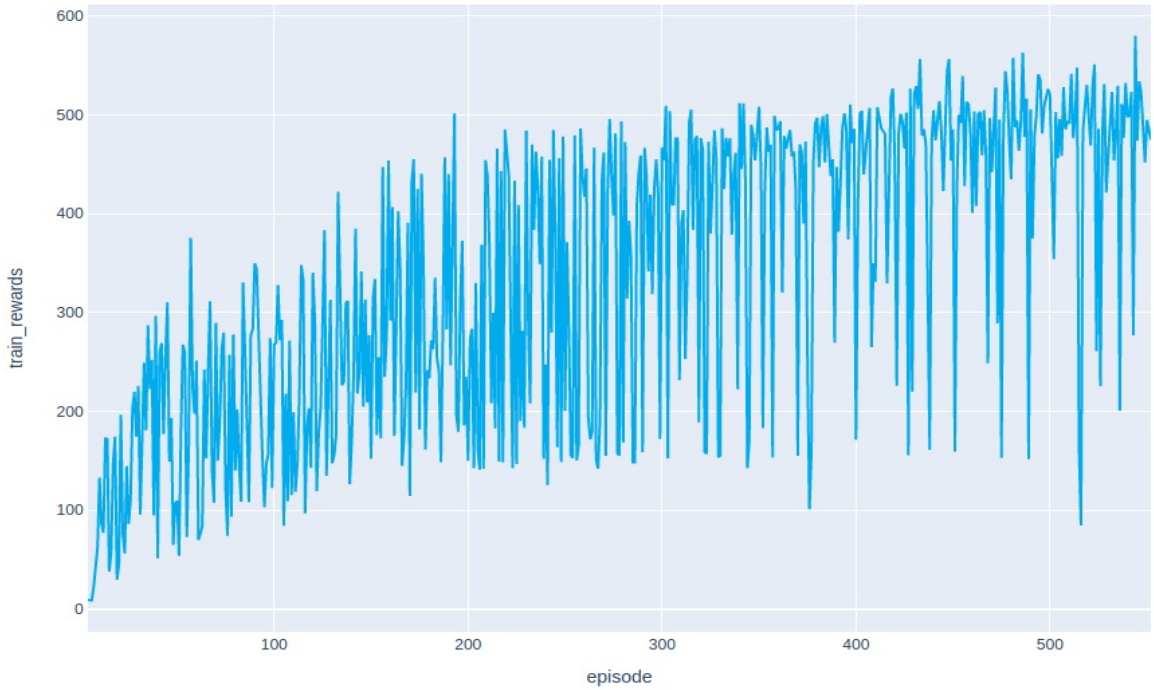



Figure 6.19: Training curve of Planet model with reward as input.

The test curve confirms that this model is worse than the original.

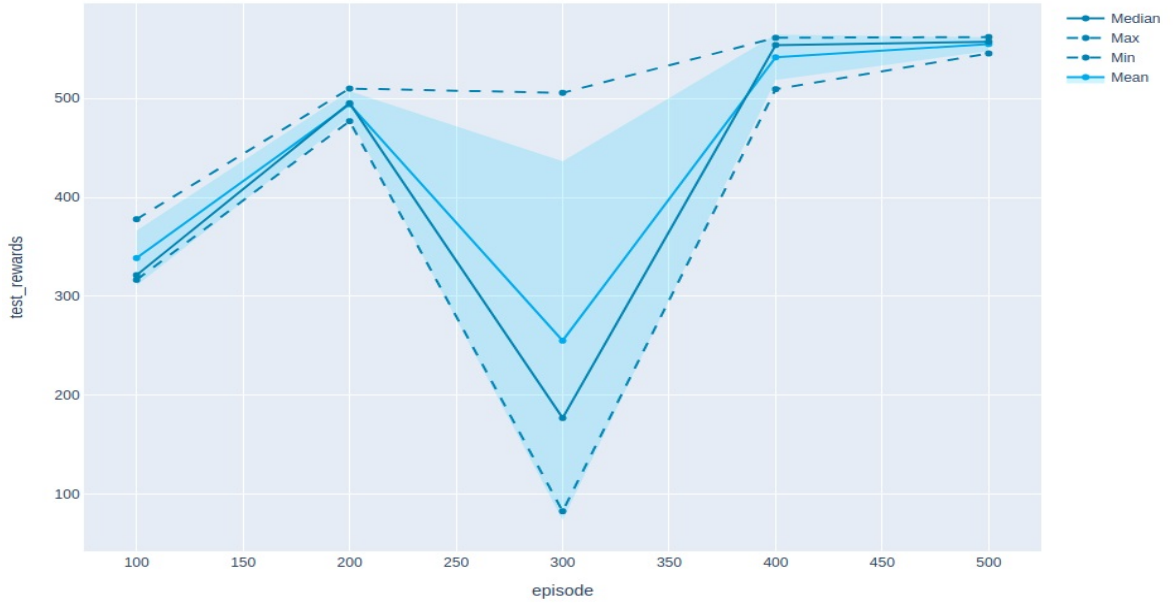



Figure 6.20: The test curve is more unstable and achieve less cumulative reward than the original model.

After 500 iterations, we saw that the model is not outperforming the original version and we stop the experiment.

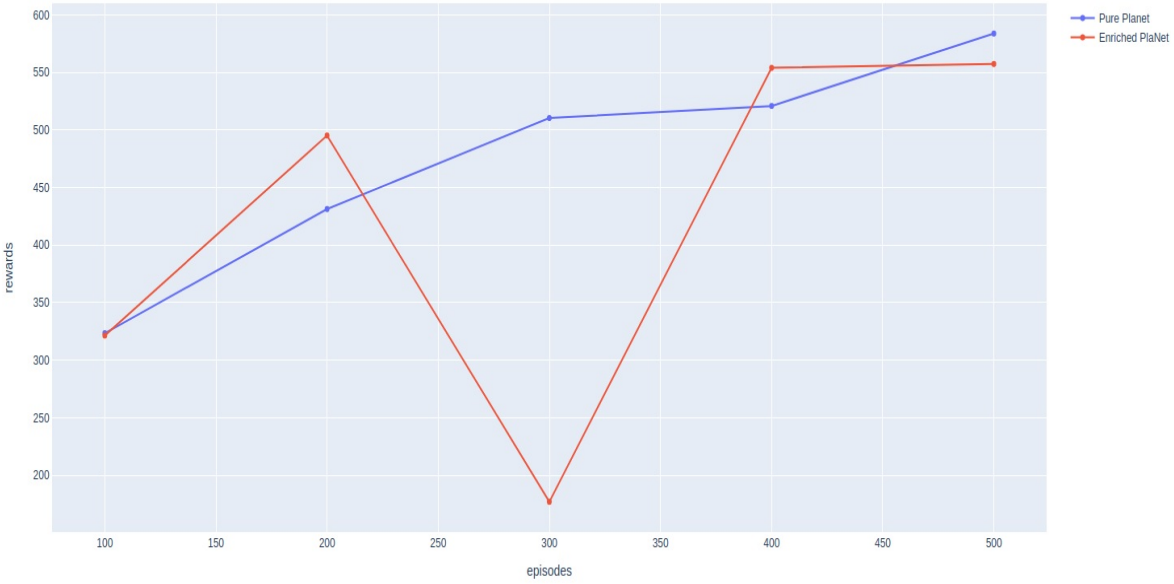



Figure 6.21: The test curve is more unstable and achieve less cumulative reward than the original model.

We tried to add the reward information to other components of the model (e.g., in the memory module or directly in the reward module), but none of these tests worked and the presented version is the one that has given the best results.

The last idea is to add regularizer to improve the model predictions. One of the main problems of using a planner in a model that is just an approximation of the real environment dynamics, is that the planner will exploit the learned model models inaccuracies. So, in the areas in which the model is uncertain, the predictions tend to be too optimistic and lead the planner to sub-optimal actions. We plot a comparison of the predicted and the rel reward obtained during the initial training episodes. From the image below we can see how initially the predictions tends to be too optimistic, and the provided plan fail to reach the expectations obtaining a low reward.

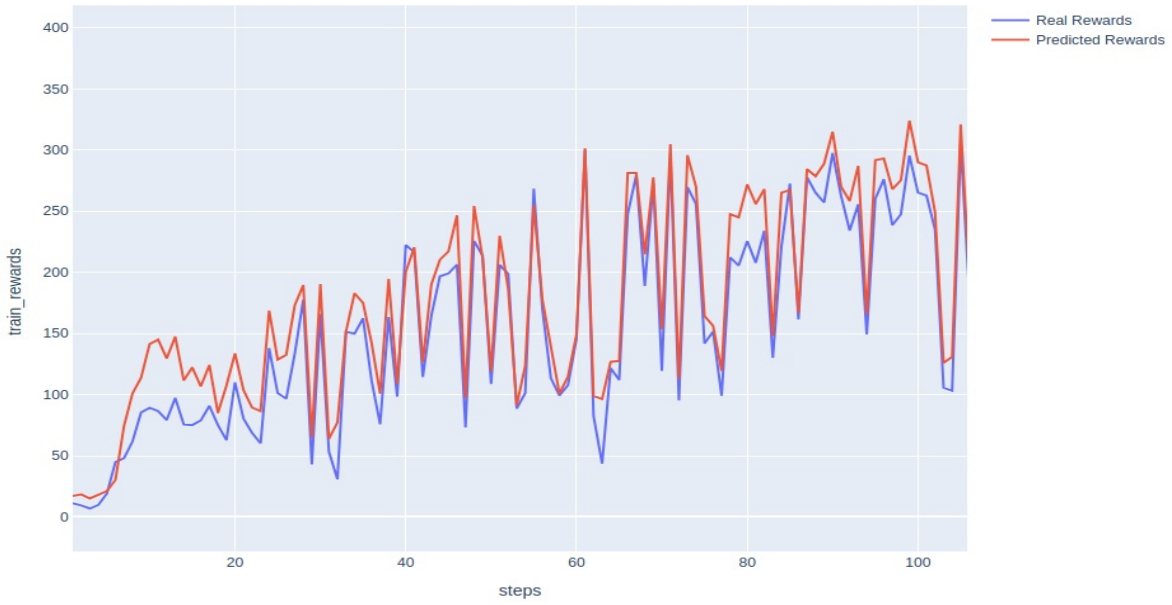



Figure 6.22: Comparison between the expected rewards predictions and the actual rewards obtained from the first 100 training episodes.

This problem is more severe in the early stages and less in later episodes when the agent has collected more data. Indeed, the more the agent interacts with the environment, the more data it collects, the more the predictions are precise. We are investigating a method to reduce this prediction gap just from the first episodes.

In their research, Rinu Boney et all [25] tries to alleviate this problem by penalizing the optimizer from considering trajectories that are outside the experience replay buffer ( that contains all the past experiences). We call this new metric: **familiarity of the trajectories**. So the planning objective is to maximize the rewards and also the familiarity of the plan respect to the data, with a new parameter $\alpha$ that modulates the weight between both costs.

$$a_t^*, \ldots, a_{t+H}^* = \underset{a_t,\ldots,a_{t+H}}{\text{argmax}} \sum_{\tau=t}^{t+H} r\left(s_\tau, a_\tau\right) + \alpha \log p\left(s_t, a_t, s_{t+1}, \ldots, s_{t+H}, a_{t+H}, s_{t+H+1}\right)$$

Where $p(o_t, a_t, ..., o_{t+H}, a_{t+H})$ is the probability of observing a given trajectory in the past experience. They approximate the joint probability of the whole trajectory as a sum of joint probabilities of each transition in the trajectory

$$a_t^*, \ldots, a_{t+H}^* = \underset{a_t,\ldots,a_{t+H}}{\text{argmax}} \sum_{\tau=t}^{t+H} \left[r\left(s_\tau, a_\tau\right) + \alpha \log p\left(s_\tau, a_\tau, s_{\tau+1}\right)\right]$$

To calculate $\log p\left(s_\tau, a_\tau, s_{\tau+1}\right)$ they uses a denoising autoencoder (DAE). DAE does not build an explicit probabilistic model $p\left(s_\tau, a_\tau, s_{\tau+1}\right)$ but learns to approximate the derivative of the log probability density. To be more specific the theory of denoising states that, for zero-mean Gaussian corruption, the optima denoising function $g(\tilde{x})$ is given by: $g(\tilde{x}) = \tilde{x} + \sigma_n^2 \frac{\partial}{\partial \tilde{x}} \log p(\tilde{x})$ where $\tilde{x}$ is the corrupted input, $p(\tilde{x})$ is the probability density function for $\tilde{x}$, $\sigma_n$ is the standard deviation of the Gaussian corruption. So given the corrupted input $\tilde{x}$ and a fully trained DAE $g(\tilde{x})$, we can derive the gradient of the log-probability of the data distribution convolved with a Gaussian distribution: $\frac{\partial}{\partial \tilde{x}} \log p(\tilde{x}) \propto g(x) - x$. They use $\frac{\partial}{\partial \tilde{x}} \log p(\tilde{x})$ instead of $\frac{\partial}{\partial \tilde{x}} \log p(x)$ assuming $\frac{\partial}{\partial \tilde{x}} \log p(\tilde{x}) \approx \frac{\partial}{\partial x} \log p(x)$.
For the experiments, they used an environment with low dimensional input (features state) and so with another model based algorithm, called PETS [26]. We initially try to replicate their solution, but the difference between the two models and the overload due to the processing of the image (they worked only with features vectors, not with frames) make the model so slow to be useless. For this reason, we use the same idea, but we implement it differently. We work directly with the prediction model and not with the planned. Another fundamental difference is that we work at training time and not at inference time. The PlaNet prediction model does not produce directly a new observation but works only in a latent space. So it makes no sense to train DAE with the observations collected in the dataset. Instead, we train the DAE directly in latent space also at training time. The second difference is about the input dimension. We feed the DAE with the entire plan at each step, so we train it by concatenating all the transitions according to the planning horizon parameter. We experiments to different ways to concatenate transitions:

1. Concatenation of triplets: every transition is composed by 3 elements: $(s_t, a_t, s_{t+1})$. But in the final concatenation, all the elements placed at the extremes will be repeated: $[(s_0, a_0, s_1), (s_1, a_1, s_2) ... (s_{10}, a_{10}, s_{11}), (s_{11}, a_{11}, s_{12})]$

2. Concatenation as chain: we remove the repetitions: $[(s_0, a_0, s_1, a_2, s_2 ... s_{10}, a_{10}, s_{11}, a_{11}, s_{12})]$

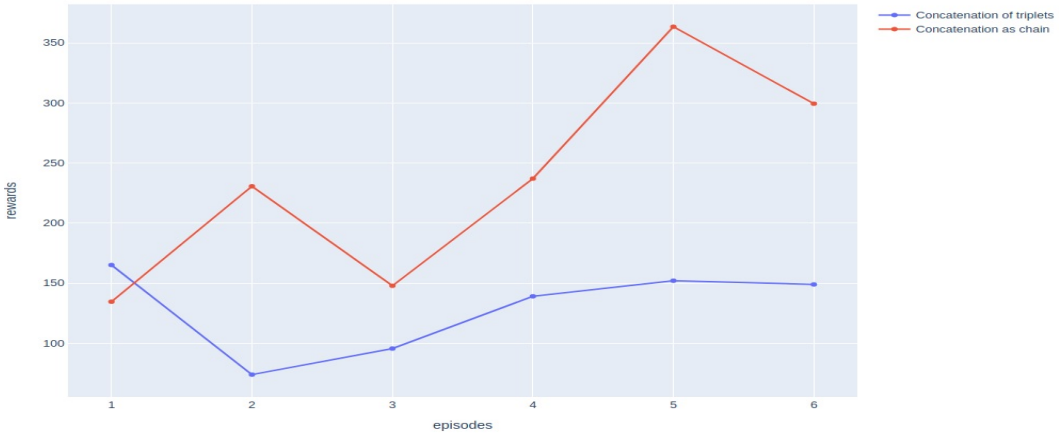



Figure 6.23: Comparison between the two strategies of input shape for the regularizer.

The final model architecture consists of one single linear layer with 600 units and a gaussian noise of zero mean and a standard deviation of 0.3. The input dimension is the sum of the belief state size, the posterior state size, and the size of the actions multiplied for the planning horizon. As we expected, the regularizer's effect is to improve the model predictions immediately and allow the model to increment the sample efficiently just from the initial episodes.


Figure 6.24: Comparison between the expected rewards predictions and the actual rewards obtained from the first 100 training episodes with regularizer.

We can see from the image above that the reward's predictions start immediately to match with the real reward when the regularizer is activated. To make more clear the comparison between the prediction of the model with and without the regularizer, we created a new plot. In this plot we indicate the difference between the predicted reward and the obtained one over the episodes of the training. We can clearly see that the prediction error of the regularized model, represented with the blue line, is clearly lower in the initial episodes and that after some episodes the two values starts to converge.


Figure 6.25: The absolute reward prediction error. The moving average technique is applied with a window of 30.

To be more clear we also calculate the relative error that remain consistent with the results above.


Figure 6.26: The relative reward prediction error. The moving average technique is applied with a window of 30.

This improvement allows the model to accumulate more rewards just in the initial episodes. This can be very useful where it is not possible to produce a huge amount of data.

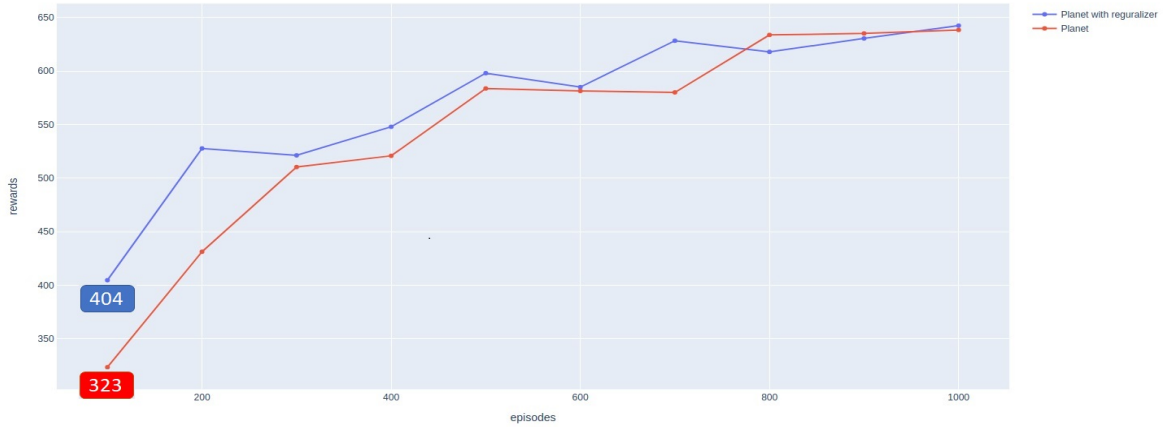



Figure 6.27: Comparison between the full trained model with and without the regularizer.

According to the research of Rinu Boney et all [25], the improvement of the regularizer decrease during the training, but in our case, the performance of the original model does not surpass the performance of the model with regularized. We think that after a certain number of iterations, the model has accumulated enough knowledge about the environment to do without the regularizer. As proof of this, we point out that to achieve these results, we have to decrease the impact of the regularizer during training. From episode 750, we deactivate it entirely. We observed an improvement, that needs to be validated with other experiments on other environments.

## 6.6 Comparisons

Now it's time to compare the model-based and the model-free approach.

We specify that for this final comparison we use the DDPG result obtained with the model trained from the features vectors, while the PlaNet results are obtained with a model trained via raw pixels.

Despite the difference in the input complexity, the model-based approach achieves better results. In particular, in the initial phase, when the model has less sample and the regularizer influence is more intense.

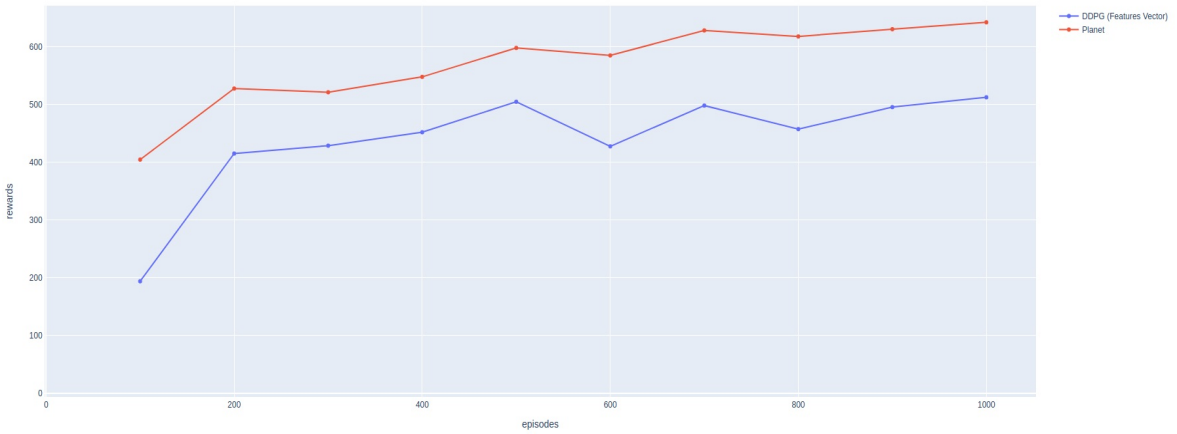



Figure 6.28: Comparison between the performance of PlaNet (trained from frames) and DDPG (trained fro features vectors) for the Ceetah environment.

We also tried other different environments, and we saw that PlaNet roughly maintain the advantage over DDPG. We specify that for these other environments, we do not use the regularizer.

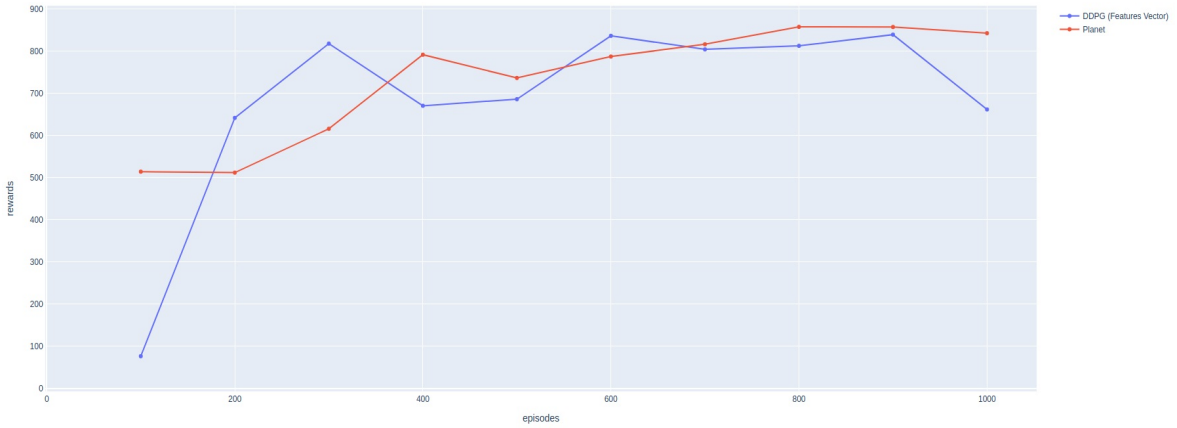



Figure 6.29: Comparison between the performance of PlaNet (trained from frames) and DDPG (trained fro features vectors) for the Cartpole-Swingup environment.

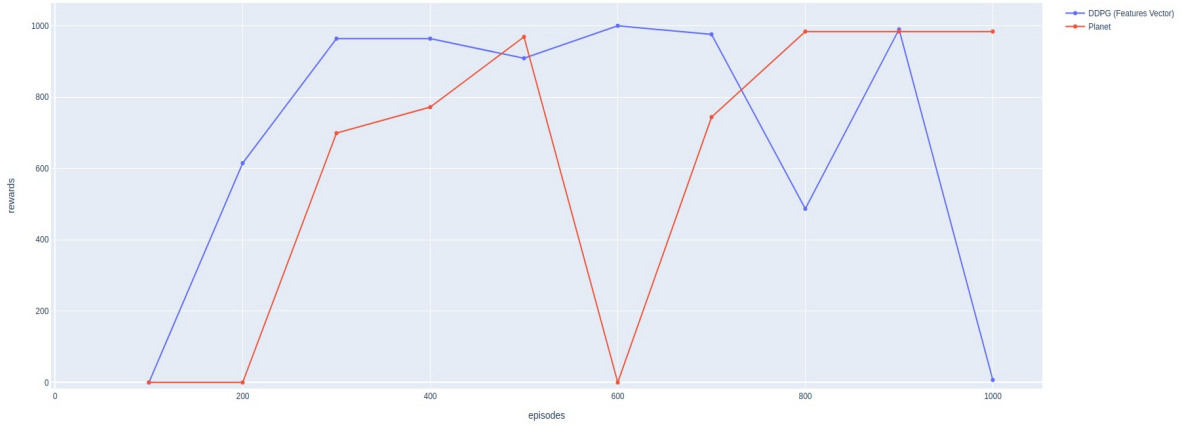



Figure 6.30: Comparison between the performance of PlaNet (trained from frames) and DDPG (trained fro features vectors) for the Reacher-easy environment.

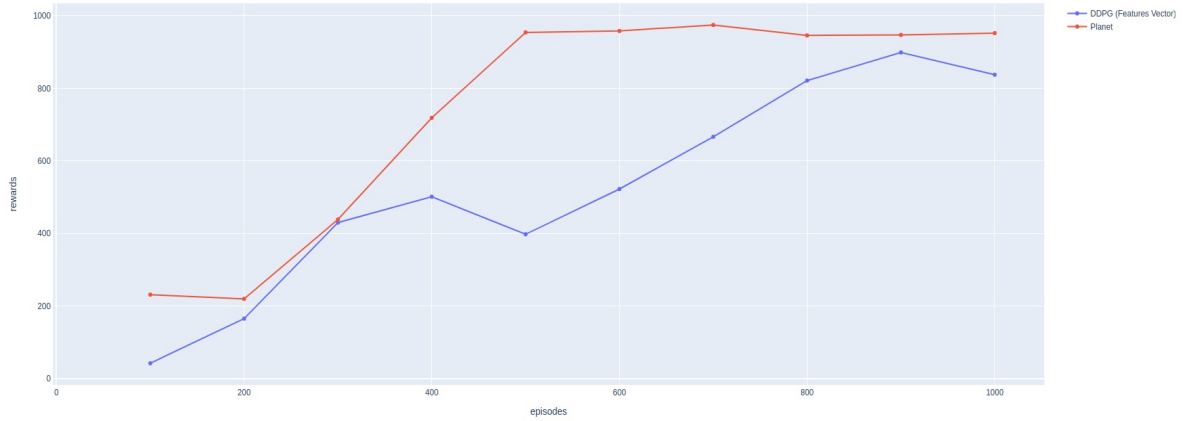



Figure 6.31: Comparison between the performance of PlaNet (trained from frames) and DDPG (trained fro features vectors) for the Walker-walk environment.

Coherently with the theory, we notice an evident difference between the training time for the model-based and model-free algorithm. The PlaNet model can achieve a better result with less sample because it makes more calculations for each step. For this reason, the training time is longer when we train a model-based agent (the same is for the amount of GPU memory). Since the model-free has not a planner, it is also faster at inference time.

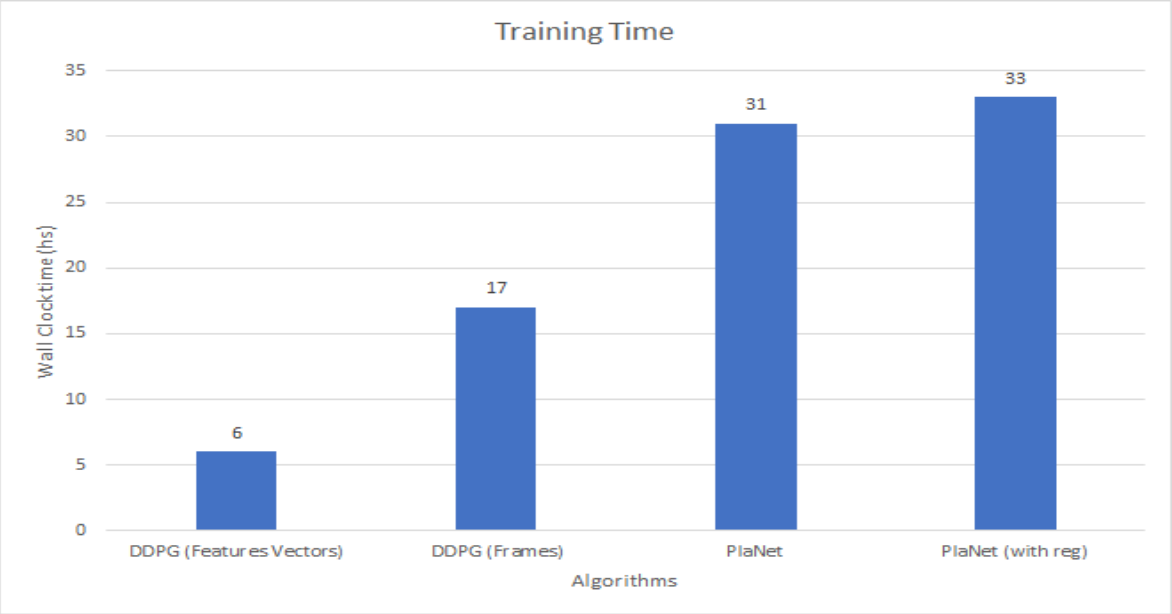



Figure 6.32: The plot shows a comparison between the training clock time (hours) required to train a DDPG linear model, DDPG convolutional model, PlaNet model and PlaNet model with the regularizer on a GPU Nvidia GeForce 1080ti..

# Chapter 7

# Conclusions

This thesis aimed to make a comparison between the model-based and the model-free approach in the context of Deep Reinforcement Learning (DRL). A set of tests have been executed over four environments from DeepMind Control Suite. The DRL algorithm known as Deep Deterministic Policy Gradient is used to test the model-free approach, while the chosen model-based algorithm was PlaNet.

The DDPG algorithm was implemented and tested over all the four environments chosen for the experiments. Some modifications, suggested by the Control Suite authors, were applied to the original algorithm and parameters to adapt them to the Control Suite environments. After this improvement, the algorithm has been able to learn a policy when a state feature was provided. These suggestions are not provided for the version with raw-pixels input. This problem is more difficult since the input is way more complicated. Features state is described by a vector of 18 dimensions (for the cheetah problem) while a single frame is a 64x64 RGB image. For the raw pixels input version the Control Suite authors have used a more advanced version of the algorithm called Distributed Distributional Deterministic Policy Gradients (D4PG). They showed that this version could also learn in this condition but is not capable of achieving the same performances of the experiments with features vector as input. Moreover, that model has required $10^8$ number of samples. For our work, we still tried to train a DDPG model with frames as input, but we have been not enough computational budget to reach such a high number of steps, so we stop the training after 1000 episodes ($10^6$ steps, like all the other experiments). These results showed how difficult is the problem of solving that benchmark environment directly from raw-pixels input.

PlaNet algorithm instead is natively designed to work with raw-pixels, and so the algorithm converged for all the tested environments. We also discovered how to improve the general performance by removing the frame's compression in the preprocessing phase asking directly the Control Suite to render the frames in the specific dimensions required.

Next, two main ideas to improve the PlaNet model were tested. The first one was about to use the obtained reward as additional information to enrich the current state, but it failed. The second idea is based on the fact that the model performances are directly connected to the reward predictions. In the early episodes, where the

model is not trained, it tends to be too optimistic and to give erroneous information to the planner. This leads to a suboptimal plan and so a low cumulated reward, because the planner will exploit the weaknesses of the predictive model instead of optimizing the real agent's behaviour. For this reason, the second idea was to improve the model prediction ability by forcing the predictions to stay close to the collected experience. In other words, during the training, we incremented the model loss when the prediction was "unlikely" with respect to the trajectory collected in the experience replay buffer. The persistence of the regularizer can penalize the model because it limits the exploration of the environment, so we reduce the impact of the it during the training. In this way, we obtained a positive impact from the first iterations, and we maintained the same performance in the last episodes. We see a positive impact from the use of the regularizer, and we believe it deserves further study and experimentation, even with other environments.

The PlaNet model was able to reach better results with respect to the DDPG algorithm even if it worked directly with raw pixels while DDPG worked with feature states. This result is confirmed also for the other three environments and showed how the model-based approach leads to better performance and more sample efficiency. Since the network architecture in the model-based approach is more complex, the training time is longer. The DDPG model is faster at inference time and required less clock time to be trained (but more samples). For the task of the train an agent in the real world, the sample efficiency is a critic parameter. In facts, the cost of acquiring samples in a real environment is an order of magnitude greater than train the model with them. Furthermore, the RGB camera is a very common, powerful and generic (not single task-specific) type of sensor that a lot of real-world robots could use. For these reasons, even if PlaNet algorithm has not already tested in a real-world scenario, we consider it a fundamental milestone to achieve the use of the Reinforcement Learning for a robot in the real world.

The following could be possible improvements for future research directions.

DDPG works well when a full markovian state is provided. We saw that the PlaNet model could produce a latent space that contains enough information to allow predictions over multiple steps. An interesting experiment could be to use PlaNet as an encoder to produce the latent states used then to train the DDPG algorithm.

In the model-based approach, the planner ability is fixed and does not improve during the training epochs, as we saw with the model-free policy. The performance improvements are due to the increment of the knowledge about the environment that allows the model to make better predictions. So, to have a better model, we need to reduce the uncertainty over the environment. An idea of improvement could be the change of the planner objective in favour of the exploration during training. Once the model is fully trained a reward exploiting objective could be restored.

# List of Figures

# Bibliography


[1] A. M. Turing, "Computing machinery and intelligence," in *Parsing the Turing Test*, pp. 23–65, Springer, 2009.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[3] P. Abbeel, A. Coates, M. Quigley, and A. Y. Ng, "An application of reinforcement learning to aerobatic helicopter flight," in *Advances in neural information processing systems*, pp. 1–8, 2007.

[4] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, *et al.*, "Dota 2 with large scale deep reinforcement learning," *arXiv preprint arXiv:1912.06680*, 2019.

[5] O. M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, *et al.*, "Learning dexterous in-hand manipulation," *The International Journal of Robotics Research*, vol. 39, no. 1, pp. 3–20, 2020.

[6] C. Finn, X. Y. Tan, Y. Duan, T. Darrell, S. Levine, and P. Abbeel, "Deep spatial autoencoders for visuomotor learning," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 512–519, IEEE, 2016.

[7] A. Nagabandi, G. Kahn, R. S. Fearing, and S. Levine, "Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 7559–7566, IEEE, 2018.

[8] Y. Tassa, Y. Doron, A. Muldal, T. Erez, Y. Li, D. d. L. Casas, D. Budden, A. Abdolmaleki, J. Merel, A. Lefrancq, *et al.*, "Deepmind control suite," *arXiv preprint arXiv:1801.00690*, 2018.

[9] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson, "Learning latent dynamics for planning from pixels," in *International Conference on Machine Learning*, pp. 2555–2565, 2019.

[10] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[11] E. Langlois, S. Zhang, G. Zhang, P. Abbeel, and J. Ba, "Benchmarking model-based reinforcement learning," *arXiv preprint arXiv:1907.02057*, 2019.

[12] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[13] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997.

[14] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," *arXiv preprint arXiv:1506.00019*, 2015.

[15] "Understanding lstm networks." `http://neuralnetworksanddeeplearning.com/chap5.html`. Posted on: 2015-08-27.

[16] M. A. Nielsen, "Neural networks and deep learning," 2018.

[17] Z. Cui, R. Ke, Z. Pu, and Y. Wang, "Deep bidirectional and unidirectional lstm recurrent neural network for network-wide traffic speed prediction," *arXiv preprint arXiv:1801.02143*, 2018.

[18] L. Jing and Y. Tian, "Self-supervised visual feature learning with deep neural networks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[19] D. P. Kingma and M. Welling, "An introduction to variational autoencoders," *arXiv preprint arXiv:1906.02691*, 2019.

[20] C. Doersch, "Tutorial on variational autoencoders," *arXiv preprint arXiv:1606.05908*, 2016.

[21] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[22] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[23] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[24] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[25] R. Boney, N. Di Palo, M. Berglund, A. Ilin, J. Kannala, A. Rasmus, and H. Valpola, "Regularizing trajectory optimization with denoising autoencoders," in *Advances in Neural Information Processing Systems*, pp. 2859–2869, 2019.

[26] K. Chua, R. Calandra, R. McAllister, and S. Levine, "Deep reinforcement learning in a handful of trials using probabilistic dynamics models," in *Advances in Neural Information Processing Systems*, pp. 4754–4765, 2018.