SAPIENZA
UNIVERSITÀ DI ROMA

Machine Learning for Social Assistive Robots

Dipartimento di Informatica, Automazione e Gestionale

Corso di Laurea Magistrale in Engineering in Computer Science

Candidate

Alessandro Lo Presti
ID number 1648663

Thesis Advisors

Prof. Luca Iocchi
Prof. Matteo Leonetti

Academic Year 2018/2019

Machine Learning for Social Assistive Robots
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: alessandro_lopresti@hotmail.it

Acknowledgments

First of all, I would like to thank Prof. Luca Iocchi for giving me the opportunity to get involved and test my skills in developing a large part of this thesis at the University of Leeds.
I would also like to thank my "british" supervisor Prof. Matteo Leonetti for guiding and supporting me during the development of this project and all the PhD students at the School of Computing - University of Leeds for having welcomed me in a manner so warm to make me feel like at home.
Secondly, I would like to thank my family who supported me throughout the cycle of studies, especially in the toughest periods and that without which I would not be the person I am today.
Finally, I would like to thank my old friends and course mates for giving me the opportunity to grow both personally and professionally.

Contents

Chapter 1

Introduction

"Machine learning research is part of research on artificial intelligence, seeking to provide knowledge to computers through data, observations and interacting with the world. That acquired knowledge allows computers to correctly generalize to new settings".

–Yoshua Bengio [10]

Because of new computing technologies, machine learning today is not like machine learning of the past. It was born from pattern recognition and the theory that computers can learn without being programmed to perform specific tasks; researchers interested in artificial intelligence wanted to see if computers could learn from data. The iterative aspect of machine learning is important because as models are exposed to new data, they are able to independently adapt. They learn from previous computations to produce reliable, repeatable decisions and results. It is a science that is not new, but one that has gained fresh momentum [33].

Nowadays, machine learning can be applied to a wide range of disciplines among which robotics stands out. In particular, it is attracting a lot of attention Socially Assistive Robotics (SAR) [39], a new field of robotics that focuses on assisting users through social rather than physical interaction. Just as a good coach or teacher can provide motivation, guidance, and support, socially assistive robots attempt to provide the appropriate emotional, cognitive, and social cues to encourage development, learning, or therapy for an individual.

Parents, teachers, and clinicians are struggling today to provide children with enough one-on-one, personalized support. Clinicians and families struggle to provide individualized educational services to children with social and cognitive deficits, whose numbers have quadrupled in the US in the last decade alone. With more than one third of children and adolescents overweight or obese, parents are struggling to provide the coaching and support needed to help children maintain healthy habits around nutrition and exercise. The thesis, as will be discussed in the following sections, takes into consideration one of these cases.

1.1 Children's Therapies Case Study

First of all, this thesis was developed through a collaboration between the DIAG department (Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti) at Sapienza University of Rome and the School of Computing at the University of Leeds. It is part of a larger project to which we have thought, more specifically, the aim of the whole project is to develop an adaptive social robot that can effectively assist therapists during children therapies, by improving social interaction between the therapist and the child.

The robot will thus help in addressing one important problem that therapists have when performing their job, which is the lack of focus and disengagement of children in the therapy. This very often results in additional work of the therapist to keep the children calm (e.g., giving him/her toys, showing videos, playing music, etc.), thus making the therapy less effective.

The environment we thought about where usually therapies are performed includes a large central area (one side is shown in the photo below roughly 100 sqm full with tools and toys where several groups are acting at a time, and a number of smaller closed rooms (about 15), where a group is acting in isolation from the others. Each group is formed by 1 therapist, 1 child (the patient) and in some cases a parent.

We identified two types of setting, namely the "floor" one, where exercises are carried out on the ground in the open space (e.g. walking or stepping through obstacles, as in photo above on the left side or play with a ball); a second setting, where the child is on a soft table (photo above on the right side) about the size of a king bed and at a height from the ground of about 1 mt, so that the therapist can act on the child in a convenient position.

A key role is the one played by the parent, which should support the whole activity by making the child comfortable and encourage and cheer him/her throughout. One specific issue that parents have to face when helping the therapist is the child cry, which is very frequent, as the child is reacting to a treatment that, although given with the greatest care and professional ability, is not a pleasure. In these situations, the main role of the parent is to distract the child and make him/her stop crying

for a better comfort of all the people involved. However, sometimes parents are not present for several reasons, including the fact that for some treatments the presence of the parent is not suitable, as it can affect the behavior of the child.
In those cases, the therapist is alone with the child and he/she has both to give the medical treatment and to make the child comfortable, actually interleaving between the two phases and thus making the medical treatment less efficient and effective.

Therapists suggested that the robot can be of great help if it will be capable of taking/complementing the role of the parent. In the case of a crying child, any action that can stop the cry and help refocusing the child on the therapeutic action will be viewed as extremely beneficial. The robot can also support the therapist to collect toys and items that are thrown on the ground by the child, especially when the action is taking place on the soft table. In addition, the robot can help positioning the objects on the floor and possibly acting as a target to be reached by the child (supported in the motion by the therapist).

Other additional relevant features considered in the whole project are:

- Use of a projector controlled by the robot (as an external actuator) that can show patterns on the floor to help and motivate children exercises. For example, the robot could control the projector to show a known scene to a child and to ask him/her to do a task embedded in that scene corresponding to the exercise to be done.
- Tele-presence for parents. As for some treatments, parents are not allowed to be with the children, the robot could act as a one-way tele-presence device showing images and sounds of the therapy to the parents without letting the children know about it. Therapists confirmed that this will be very helpful for parents to increase their understanding of the therapy.
- Generation of a report describing the activities performed during the therapy. Therapists have little time to write down a report about each therapy and such reports are often not very detailed. The robot can help in collecting information (e.g., data and videos) about the therapy session and organize them to improve the quality of the reports. In particular it may be interesting to place the camera on the robot arm to take a good view of the scene considering that the people involved keep changing their position in the environment.

Of the whole project, the thesis actually develops an agent capable of analyzing the environment surrounding it through the study of features chosen by us in the design phase (thus understanding what state it is in) and acting accordingly by performing speech actions based on the policy learned during the training phase. The agent is represented through a multi-thread application developed in Python [36] that can be run through any computer meeting the requirements in the project's requirements.txt file [17] and has a webcam installed.

Due to the lack of privacy permits we have not been able to record video and audio in the hospitals containing the faces and voices of children and physiotherapists, as well as not having children available to actually evaluate our application in the testing phase. For this reason in the training phase we decided to consider the faces and voices of adults using datasets that we will discuss later, then, in the testing phase, we employed the PhD students present in the laboratory of the School of Computing of the University of Leeds.

1.2 Thesis Overview

The project is structured as follows:

- Supervised Learning module: in this section the agent analyses videos and audio clips acquired using its sensors (camera and microphone) to perform classification tasks, that is, assigning some values to the features characterizing the set of states of our Markov Decision Process. All these activities are related to the recognition of emotions experienced by the patient at each time interval. For sake of simplicity we have assumed that the images will represent only the child/person being treated and an object possibly used to calm them. For all these activities we used already trained Artificial Neural Networks.

- Reinforcement Learning module: we have formulated the problem under a Reinforcement Learning point of view, that is, the goal of the agent is staying as much as possible in states in which the child/person is emotionally well and relates correctly to the object considered. This potentially results in not slowing down the therapy and engaging the physiotherapist in calming the child/person. Therefore, we modelled an MDP considering each state made up of some features/state variables that we will describe later.
  A crucial point on which the whole thesis is based concerns the importance of the features characterizing the state the agent perceives. In fact we have developed our own learning algorithm based on the Decision Tree Regressor able to show us which features are more or less important. In this way we have been able to retrain our model by discarding the less important features and compare the performances obtained by analyzing whether, indeed, it is possible to reduce the complexity of the problem we have set ourselves to solve.

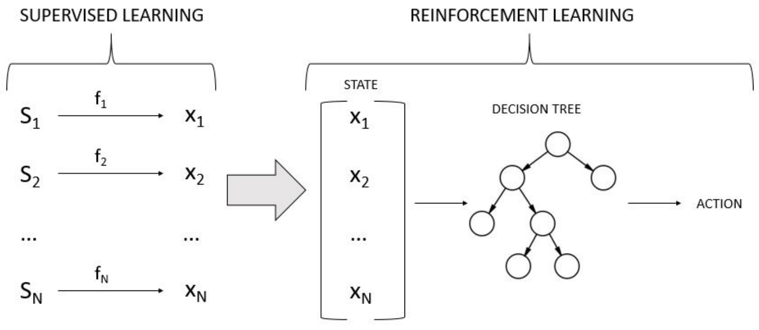


Figure 1.1. Overview of the thesis framework

Therefore, summarizing, each $\mathrm{S_i}$ for $i = 1, ..., N$ represents a sensor with which the agent acquires raw data and passing it to a Neural Network, represented as a function $\mathrm{f_i}$ for $i = 1, ..., N$, it gets the value of the i-th state variable. After computing the values of all features, they are grouped all together to form the actual state that will be passed to the trained Decision Tree Regressor returning the corresponding action to execute.

1.3 Document Structure

The following chapters are structured as follows:

- In Chapter 2, we will discuss under a mathematical point of view the Neural Network models from which we were inspired to subsequently classify raw data into values of state variables as well as the basic algorithms for Decision Tree in order to proper understand the implementation of our algorithm in a Reinforcement Learning setting.
- In Chapter 3, we will define the basics of Reinforcement Learning such as what a Markov Decision Process is and how to learn a policy describing what the difference between Off/On-policy algorithms is. Furthermore, regarding the current research, we will explain how learning takes place in a reinforcement learning problem using neural networks (Deep-Q-learning) as well as explaining how one of the most widely used algorithms nowadays (Proximal Policy Optimization) works.
- In Chapter 4, we will define our MDP describing our features and we will explain our learning model using Dynamic Decision Trees, that is, at each iteration we build a new tree that can learn a better policy until we reach an approximation of the optimal policy. Moreover, we will consider the feature importance, the core of this thesis, explaining how we decided to combine feature importances returned by multiple decision trees in order to actually understand which are the most important features in the problem we considered.
- In Chapter 5 we will show the Environment Simulator we developed and the experiments we have carried out, especially regarding the change of parameters and what results are actually obtained by discarding some of the features according to the feature importance metrics.
- Finally, in Chapter 6, we will briefly describe what could be the future implementations following the results of this thesis and considering the ideas of the whole project.

Chapter 2

Supervised Learning for Classification and Regression

2.1 Feedforward Neural Network

A Feedforward Neural Network (FNN) defines a mapping $f(x;\theta)$ and it adaptively learns the parameters θ to best approximate a target function f^* with respect a loss measure. It has a network structure because its internal operations can be represented through a direct acyclic graph. It is called neural because it is inspired by the human brain. In detail, a neural network is constituted of stacked layers, each of them made of multiple neurons. The first one is called input layer; the last one is the output layer; in the middle there are hidden layers.

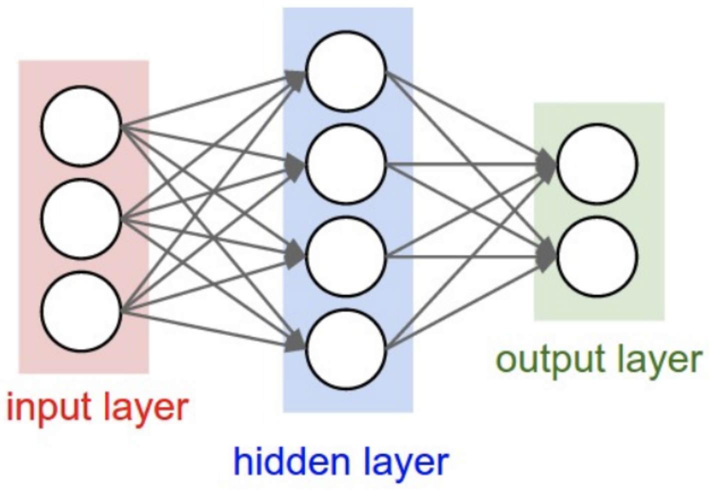


Figure 2.1. A neural network with 3 neurons in input, 4 in the fully connected layer, and 2 in the output [11]

For regular neural networks, the most common layer type is the fully-connected layer in which neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections (Fig. 2.1).

If we take into consideration a single neuron in the hidden layer of Fig. 2.1, and we define:

1. x: the neuron's output
2. w_i: the weights associated to its connections with the previous layer
3. σ: the activation function

we have that $x = \sigma(\sum_i w_i \cdot x_i)$.

Neural Networks with a linear output layer and at least one hidden layer with any "squashing" activation function (e.g. sigmoid) are demonstrated to be universal approximators, that is, they can approximate any Borel measurable function with any desired amount of error, provided that enough hidden units are used. It works also for other activation functions (e.g., ReLU) that we will describe later. This is a powerful results but it does not tell how to build that hidden layer neither how to train the network and this is why in general we need to stack multiple layers.

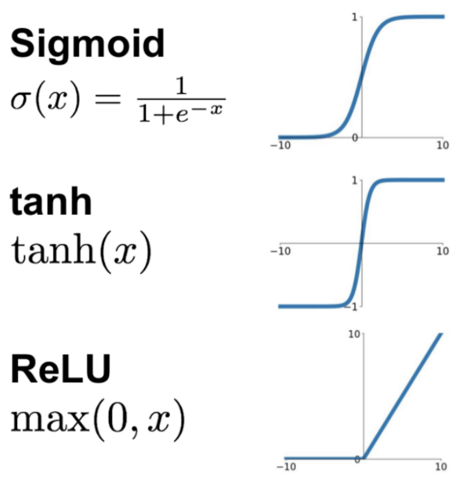


Figure 2.2. Activation functions

As illustrated in the formulas above, the output of a neuron depends on the activation function. Usually, the latter is a non linear function that gives the model the ability to approximate functions that are not linear or to classify non linearly separable data. The main types of activation functions are:

1. $Sigmoid(x) = \frac{1}{1+e^{-x}}$. It squashes the input to the range: $[0, 1]$ and since it can be interpreted as a probability measure it is often used in the output layer. The first drawback is that the gradient of points near to 0 or 1, is close to zero causing the vanishing gradient problem. Secondly, the output is always positive and this results in gradients that are always positive or negative for the weights of the layer. For example if at some layer we have a 2-dimensional output: x_1, x_2 that is positive because it is the result of a *sigmoid* and a loss function: L, that depends on $f = w_1x_1 + w_2x_2$, we have that $\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial w_i} = \frac{\partial L}{\partial f}x_i$ that

has the same sign of $\frac{\partial L}{\partial f}$. This means w_i needs to be both updated of the same positive of negative values, determining a slower convergence and the zig-zag update dynamics.

2. $Tanh(x)$. It is bound to the range (-1, 1) and, like sigmoid, it suffers from the vanishing gradient problem, but since it is zero centered the convergence is faster. In practice, optimization is easier in this method hence it is always preferred over *sigmoid* function. And it is also common to use the $Tanh$ function in a state to state transition model such as recurrent neural networks.

3. $ReLU(x) = max(0, x)$. It does not suffer from vanishing gradient since its derivative does not saturate. It is quicker to compute because it does not require exponential functions. However it suffers from the dying neurons: when a weight is updated after a great gradient and becomes a big negative number, it will outputs only zero values since that weight will difficult be updated for a large positive value.

2.1.1 Weighted Loss Function

Neural networks cast the task of learning as an optimization problem and solve it through the application of specific algorithms. In particular we define distinctively a loss function that measure the error and the optimizer to minimize it. The latter can be Stochastic Gradient Descent (SGD), Adaptive Moment Estimation (Adam) and others, but in the following we will only talk about the loss function.

Formally, the objective function measure the deviation between the predicted value $\hat{y}$ and the true label y. If we have a dataset $D = \{x_i, y_i\}^N$, the loss function $\mathcal{L}$ is defined as: $\mathcal{L}(\theta) = \frac{1}{N}\sum_i^N L(y_i, f(x_i, \theta))$, where θ are the model's learnable parameters and f is the neural network.

The choice of network output units and cost function are related, their combination depends on the kind of machine learning problem we are facing.
More in detail:

- Regression: linear units with an identity activation function $y = W^T h + b$ and maximum likelihood (cross-entropy) as cost function are used together in regression problems. If we assume a Gaussian distribution noise model $P(t|x) = \mathcal{N}(t|y, \beta^{-1})$ then the maximum likelihood estimation corresponds to mean squared error $\mathcal{L}(\theta) = \frac{1}{2}(t - f(x, \theta))^2$.

- Binary Classification: we consider output unit using sigmoid activation function because in this kind of problem the output must be between 0 and 1 and sigmoid behaves in such a way. The likelihood corresponds to a Bernoulli distribution: $\mathcal{L}(\theta) = -lnP(t|x) = -ln\sigma(\alpha)^t(1-\sigma(\alpha))^{1-t} = -ln\sigma((2t-1)\alpha) = softplus((1-2t)\alpha)$ with $\alpha = w^t h + b$, thus we consider the binary cross-entropy as loss function.

- Multi-class classification: we consider the softmax activation function
$y_i = softmax(\alpha)_i$ and the categorical cross-entropy $\mathcal{L}(\theta)_i = -lnsoftmax(\alpha)_i$

2.1.2 Learning algorithms

In a neural network information flows forward through the network when computing network output y from input x. To train the network we first need to compute the gradients with respect to the network parameters θ. The backpropagation algorithm works by computing the gradient of the loss function with respect to each weight by the chain rule, iterating backwards one layer at a time from the last layer to avoid redundant calculations of intermediate terms in the chain rule. It is important to remark that backpropagation is only used to compute the gradients and it is not a training algorithm, furthermore, it is not specific to FNNs but it is used by other neural network architectures as well.

Once we have the gradient, to change the parameters of the model and actually start the learning process we can use one of the following learning algorithms:

- Stochastic Gradient Descent (SGD): at each iteration a subset (minibatch) $x^{(1)}, ..., x^{(m)}$ of m examples is sampled from the dataset D. The gradient is computed through the formula: $g = \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)}; \theta^{(k)}), t^{(i)})$ and the update is applied $\theta^{(k+1)} = \theta^{(k)} - \eta g$ where k refers the k-th iteration and η represents the learning rate.

- SGD with momentum: it can accelerate the learning with respect to the first algorithm since we consider the momentum μ, that is, how strong should be the velocity w.r.t. the gradient. Therefore, after computing the gradient estimate g, the algorithm computes the velocity $v^{k+1} = \mu v^{(k)} - \eta g$ with $\mu \in [0, 1)$ and applies the update $\theta^{(k+1)} = \theta^{(k)} + v^{(k+1)}$.

- Algorithms with adaptive learning rates: they have an adaptive learning rate that does not change overtime of a fixed value, in fact, at any step of the algorithm, it is possible to determine whether the learning rate should be increased or decreased. We do not go into details but some examples are AdaGrad, RMSProp and Adam.

2.2 Convolutional Neural Network

Convolutional Neural Networks (CNN) is a deep model that performs well with a variety of tasks such as image classification, natural language processing, and signal processing. CNNs are explicitly designed to deal with multi-dimensional input and overcome the high number of parameters that are requested by standard FNN. For example, a single RGB image of size 64x64, in an FNN would require: $64 \cdot 64 \cdot 3 = 12288$ neurons as input. The issues that arise when the FNN is over parametrized are the following:

1. a huge number of input neurons will require more layers at a high computation cost and time required for training.

2. over-parametrization is a symptom of overfitting: in the specific case of an image, the FNN would behave too meticulous since it will take into account each single pixel.

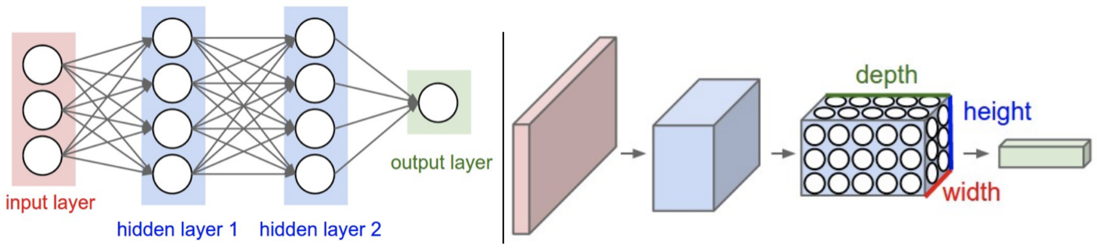


Figure 2.3. Comparison of FNN versus CNN. In CNN each layer has 3 dimension: depth, height, length.[11]

In order to take into account the multidimensional input, CNN's neurons are organized in three dimensions and to reduce the overall complexity, the neurons in each layer are connected only to a portion of the previous one. This is the opposite of what happens in fully connected neural networks. The main types of layers used in CNN are:

1. convolutional: consists of groups of neurons apply a scalar product with the connecting portions of the input.
2. pooling: downsampling to reduce the dimensionally
3. fully connected: produces the final predictions. Generally, it is preceded by a flattening operation since the convolutional outputs are always 3-dimensional

2.2.1 Convolutional Layer

The convolutional layer is the building block of CNNs. It consists of a set of filters (or kernel), whose parameters are learned during the training. A single filter is a multidimensional matrix with height, length and the same width of the input layer. For example, if we have an RGB image of size: $64 \cdot 64 \cdot 3$ and filter of size: $6 \cdot 6 \cdot 3$, the total number of learnable parameters of the latter are: $6 \cdot 6 \cdot 3 = 108$. It is immediate to note the crucial reduction of parameters with respect to a classical feed-forward neural network, that would have required $64 \cdot 64 \cdot 3 = 12288$ weights in the input layer.

During forward pass, each filter is shifted through the input and performs a convolution that in practice is a simple dot product. The filter can be interpreted as a particular feature and the convolution asses how much that feature is present in the portion of the input. During the training, CNN learns the weights of such filters, so that they can extract the actual features to discriminate the final prediction. In the end, each filter outputs a 2-dimensional matrix that for what we said before it is called feature map.

As we said before, each filter has a limited area called receptive field that is connected only to a small portion of the input to reduce the number of parameters. This technique as the name suggests is called parameter sharing and is based on the fact the same feature is present in multiple regions of the input: for example if we want to detect the border of an image, we could use a filter that properly activates when

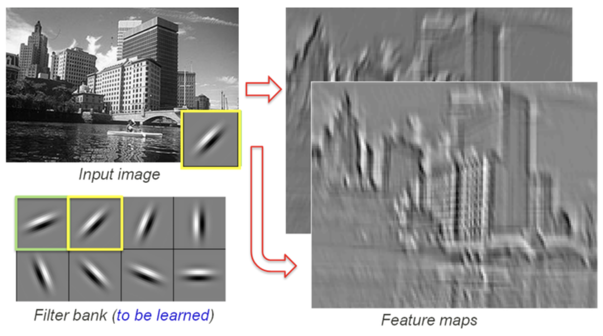


Figure 2.4. 2 different filters are applied to the input image, resulting in 2 different feature maps [34]

it is convoluted with them. In any case, the filter is applied to all the input and at the end of any convolutional layer is always present an activation function that is generally a ReLU. The parameters of a convolutional layer are:

1. spatial extent of the filter (F)
2. depth (K): the number of filters that determines the depth of the output
3. stride (S): the number of pixels to skip every time the filter is convoluted
4. zero-padding: consists of adding some zeros to the image border to have control on the output dimension. Specifically if $P = 1$ each border has one zero and so on.

With the notation above, we can deduct the output dimension of a convolutional layer. Given an image of size: $W_1 \cdot H_1 \cdot D_1$, the output is:

1. $W_2 = 1 + (W_1 - F - 2P)/S$
2. $H_2 = 1 + (H_1 - F - 2P)/S$
3. $D_2 = K$

The number of parameters for a single filter are: $F \cdot F \cdot D_1$ and globally: $F \cdot F \cdot D_1 \cdot K$

2.2.2 Pooling Layer

The pooling layer is used to reduce the dimensionality of the given output's layer and consequently to obtain a reduced number of parameters at a less computational cost. Generally, it has a local extension of 2x2 $(F = 2)$ and it outputs the maximum in the region it is applied (max pooling) with a given stride: S. It performs a downsampling only through the width and height: the original depth is left unchanged. Also in this case it is possible to define the output dimension as:

1. $W_2 = 1 + (W_1 - F)/S$
2. $H_2 = 1 + (H_1 - F)/S$
3. $D_2 = D_1$

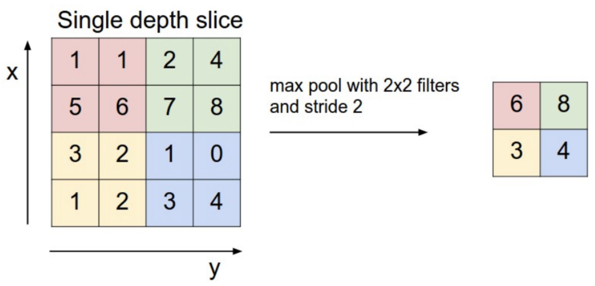


Figure 2.5. Max pool with stride 2.(Left: original input, Right: pooling output) [11]

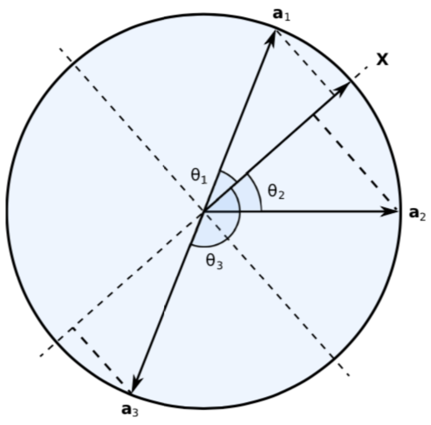


Figure 2.6. An input vector $\boldsymbol{x}$ with three anchor vectors: $\boldsymbol{a_1}, \boldsymbol{a_2}, \boldsymbol{a_3}$, in the sphere S: [15]

2.2.3 Mathematical Interpretation

In this section, we will describe CNNs from a geometric point of view, so that it is possible to understand the necessity of an activation function at the end of each layer and the advantages of multiple stacked convolutional layers. In [15], the CNN is interpreted as a RECOS model: set of operation that performs Rectified Operation On a Spere. Specifically, each filter is interpreted as an anchor vector in a sphere, because they serve as reference signals. Each convolution between a local region and filter is translated as a similarity score between the anchor vector and the vector representation of that region.

In the RECOS model, a negative correlation is set to zero, and the reason will be described later. Given a image's region in a vectorial form: $x = (x_1, \ldots, x_N)^T$, and a sphere centered in the origin of unitary size: $S = \{\boldsymbol{x} \mid ||x|| = 1\}$, we analyze the geodesic distance between x and an anchor vector: $\boldsymbol{a_1}$.

In particular, the geodesic similarity: d is the output of the convolutional step between a filter and the portion of input it is applied, that is a simple dot product. Specifically, $d = \boldsymbol{a_1}^T\boldsymbol{x} = cos(\theta)$ where θ is the angle between the two vectors (Fig. 2.6). The longer the correlation, the shorter the distance. The problems with this similarity score, happen when $\theta > 90$, because $cos(\theta) < 0$. Even if we could state a negative value correctly indicates high distances, issues arise when an additional anchor vector is negatively convoluted on the previous output producing a positive outcome. In this case, low correlated anchor vectors have outputted a positive score. Similarly, the model is not able to distinguish with two anchor vectors, between positive/negative response and vice-versa. To solve this problem, we apply a Rectified Linear Unit (ReLU) so that negative correlation is set unambiguously to zero.

The paper ([15]), further discuss the need to multiple convolutional layers. The point is that each layer operates at different pattern level detection: from more general to more specific patterns. For example, the paper tested on MNIST dataset that by changing the background's images, the first layer's filters change while the second layer is unaltered. This is due to the first layer is more general and capture the general picture so also the background, while the following ones tend to concentrate on more specific invariant features.

Finally, we will conclude with the role of the fully connected layers. Generally, the convolutional layers perform feature selection, while the last dense layers are used to detect the surface boundary required for the final classification. In details, the latter need to cluster the input vector to the specified dimension. For example with an FC layer of size 100 and 4, the features are first clustered to 100 clusters and then to 4.

2.3 Regression with Decision Trees

Now, we will describe the main decision tree models for regression problems since in our approach, that we will explain in next chapters, we were inspired by them.

2.3.1 ID3

The ID3 algorithm is considered as a very simple decision tree algorithm [25][31]. The ID3 algorithm is a decision-tree building algorithm. It builds a decision tree for the given data in a top-down fashion, starting from a set of objects and a specification of properties. At each node of the tree, one property is tested based on maximizing information gain and minimizing entropy, and the results are used to split the object set.

This process is recursively done until the set in a given sub-tree is homogeneous (i.e. it contains objects belonging to the same category). This becomes a leaf node of the decision tree [22].The ID3 algorithm uses a greedy search. It selects a test using the information gain criterion, and then never explores the possibility of alternate choices.
This makes it a very efficient algorithm, in terms of processing time.

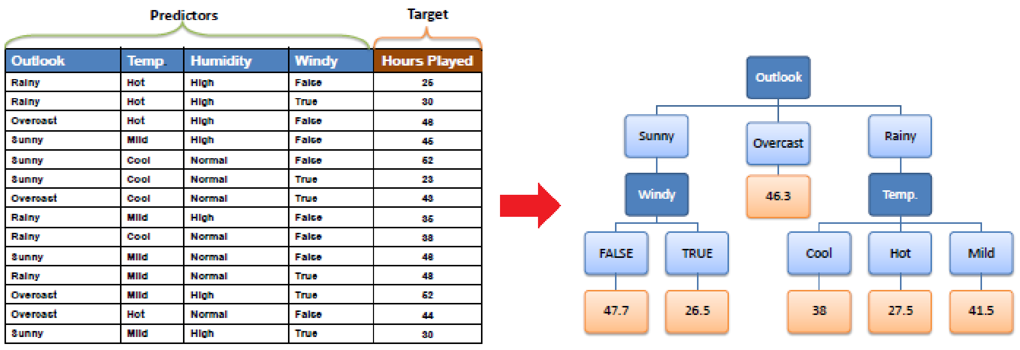
The ID3 algorithm just explained refers to a classification task, however, it can be used to construct a decision tree for regression by replacing Information Gain with Standard Deviation Reduction. The Standard Deviation Reduction is based on the decrease in standard deviation after a dataset is split on an attribute.

It has the following advantages:

- Understandable prediction rules are created from the training data.
- Builds the fastest tree.
- Builds a short tree.
- Only need to test enough attributes until all data is classified.
- Finding leaf nodes enables test data to be pruned, reducing number of tests.
- Whole dataset is searched to create tree.

And the following disadvantages:

- Data may be over-fitted or over-classified, if a small sample is tested.
- Only one attribute at a time is tested for making a decision.
- Does not handle numeric attributes and missing values.



Predictors				Target
Outlook	Temp.	Humidity	Windy	Hours Played
Rainy	Hot	High	False	25
Rainy	Hot	High	True	30
Overcast	Hot	High	False	46
Sunny	Mild	High	False	45
Sunny	Cool	Normal	False	52
Sunny	Cool	Normal	True	23
Overcast	Cool	Normal	True	43
Rainy	Mild	High	False	35
Rainy	Cool	Normal	False	38
Sunny	Mild	Normal	False	46
Rainy	Mild	Normal	True	48
Overcast	Mild	High	True	52
Overcast	Hot	Normal	False	44
Sunny	Mild	High	True	30

Figure 2.7. Resulting regression decision tree considering the dataset on the left side

2.3.2 CART

CART stands for Classification and Regression Trees [5].It is characterized by the fact that it constructs binary trees, namely each internal node has exactly two outgoing edges. The splits are selected using the twoing criteria and the obtained tree is pruned by cost–complexity Pruning. When provided, CART can consider misclassification costs in the tree induction. It also enables users to provide prior probability distribution. An important feature of CART is its ability to generate regression trees. Regression trees are trees where their leaves predict a real number and not a class. In case of regression, CART looks for splits that minimize the prediction squared error (the least–squared deviation). The prediction in each leaf is based on the weighted mean for node [1].

It has the following advantages:

- CART can easily handle both numerical and categorical variables.
- CART algorithm will itself identify the most significant variables and eliminate non significant ones.
- CART can easily handle outliers.

And the following disadvantages:

- CART may have unstable decision tree. Insignificant modification of learning sample such as eliminating several observations and cause changes in decision tree: increase or decrease of tree complexity, changes in splitting variables and values.
- CART splits only by one variable.

2.3.3 C4.5

C4.5 is an evolution of ID3, presented by the same author [26]. The C4.5 algorithm generates a decision tree for the given data by recursively splitting that data. The decision tree grows using Depth-first strategy. The C4.5 algorithm considers all the possible tests that can split the data and selects a test that gives the best information gain (i.e. highest gain ratio). This test removes ID3's bias in favor of wide decision trees [38]. For each discrete attribute, one test is used to produce many outcomes as the number of distinct values of the attribute. For each continuous attribute, the data is sorted, and the entropy gain is calculated based on binary cuts on each distinct value in one scan of the sorted data. This process is repeated for all continuous attributes. The C4.5 algorithm allows pruning of the resulting decision trees. This increases the error rates on the training data, but importantly, decreases the error rates on the unseen testing data. The C4.5 algorithm can also deal with numeric attributes, missing values, and noisy data. It has the following advantages [18]:

- C4.5 can handle both continuous and discrete attributes. In order to handle continuous attributes, it creates a threshold and then splits the list into those whose attribute value is above the threshold and those that are less than or equal to it [38].
- C4.5 allows attribute values to be marked as? For missing. Missing attribute values are simply not used in gain and entropy calculations.
- C4.5 goes back through the tree once it's been created and attempts to remove branches that do not help by replacing them with leaf nodes.

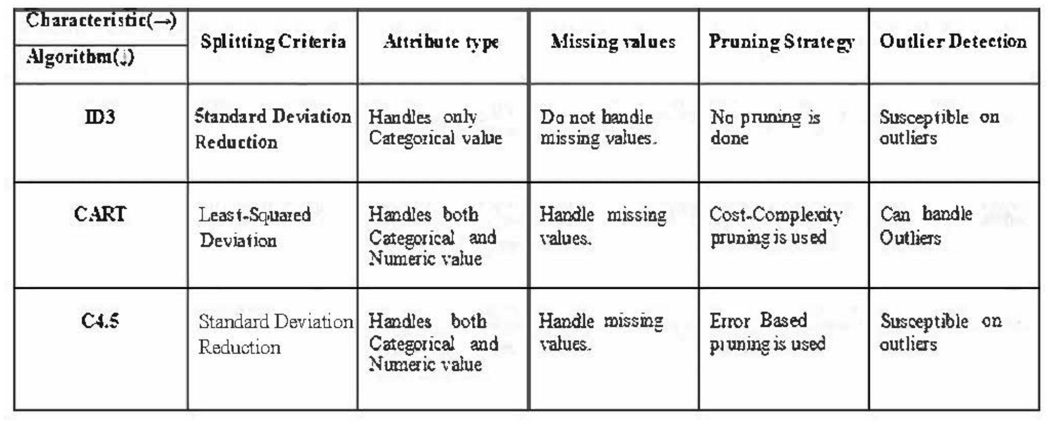
And the following disadvantages:

- C4.5 constructs empty branches; it is the most crucial step for rule generation in C4.5. We have found many nodes with zero values or close to zero values. These values neither contribute to generate rules nor help to construct any class for classification task. Rather it makes the tree bigger and more complex [18].

- Over fitting happens when algorithm model picks up data with uncommon characteristics. Generally C4.5 algorithm constructs trees and grows it branches 'just deep enough to perfectly classify the training examples'. This strategy performs well with noise free data. But most of the time this approach over fits the training examples with noisy data. Currently there are two approaches are widely using to bypass this over-fitting in decision tree learning [18].
- Susceptible to noise.

2.3.4 Summary

The basic characteristics for regression of the above three algorithms are explained in the table below:



Characteristic(→) / Algorithm(↓)	Splitting Criteria	Attribute type	Missing values	Pruning Strategy	Outlier Detection
ID3	Standard Deviation Reduction	Handles only Categorical value	Do not handle missing values.	No pruning is done	Susceptible on outliers
CART	Least-Squared Deviation	Handles both Categorical and Numeric value	Handle missing values.	Cost-Complexity pruning is used	Can handle Outliers
C4.5	Standard Deviation Reduction	Handles both Categorical and Numeric value	Handle missing values.	Error Based pruning is used	Susceptible on outliers

Figure 2.8. basic characteristic of decision tree algorithms.

Chapter 3

Reinforcement Learning for Decision-Making

Now we will explain the basic concepts of reinforcement learning and we will continue explaining the importance of neural networks in these tasks, concluding everything with one of the most currently used and studied algorithms.
The first part of the chapter corresponds to concepts that were studied during the degree course at Sapienza University of Rome and that we were able to apply during this project while the part related to deep reinforcement learning was studied in depth at the University of Leeds through the development of applications in environments such as OpenAI Gym. All this chapter will be mainly based on information provided by [12][35].

3.1 Definition of Markov Decision Process

In Reinforcement Learning we want to learn a function not only with respect to features as in Supervised Learning but also with respect to time.
This is why we consider a dynamic system, more in detail, as represented in Fig. 3.1 the classical view of a dynamic system is:

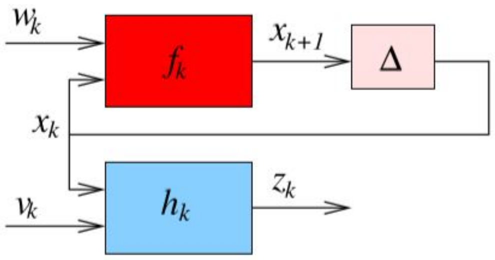


Figure 3.1. Classical view of a dynamic system

where:

- x is the state, that is, any way of representing the information of dynamic system. The evolution is given by a sequence of state.

- f is the transition function, it models how states evolve overtime, thus how to compute the next state given the current state.
- w, v is noise, that is, whatever is affecting the behavior of the system and it is not modeled in the state.
- z represents observations, sometimes the state can be observable and other times not therefore we need some sensors/devices to observe some information about the system. In general we have an observation model h that extract information from the state, the outcome of h are some observations denoted by z.
- h is the observation model

Learning in dynamic systems means we do not know f and h thus we are not able to predict the future, however, what we can do is trying, that is, we execute an action and we move to the next state gaining knowledge and according to that we can do better next time.
The state x encodes:

- all the past knowledge needed to predict the future
- the knowledge gathered through operation
- the knowledge needed to pursue the goal

Some examples are the configuration of a board game or a robot device and the screenshot of a video-game.
A dynamic system is actually represented as:

- X: set of states
  - explicit discrete and finite representation $X = \{x_1, ..., x_n\}$.
  - continuous representation $X = F(...)$ (state function.
  - probabilistic representation $P(X)$ (probabilistic state function)
- A: set of actions
  - explicit discrete and finite representation $A = \{a_1, ..., a_m\}$
  - continuous representation $A = U(...)$ (control function)
- δ transition function
  - deterministic/non-deterministic/probabilistic
- Z: set of observations
  - explicit discrete and finite representation $Z = \{z_1, ..., z_k\}$
  - continuous representation $Z = Z(...)$ (observation function)
  - probabilistic representation $P(Z)$ (probabilistic observation function)

Therefore, considering a dynamic system representation, a Markov Decision Process is a process that has the Markov property.

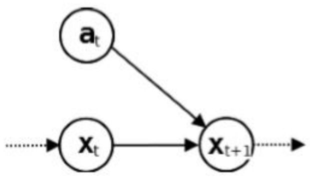


Figure 3.2. graphical model of an MDP

3.1.1 Markov Property

According to the Markov Property:

- Once the current state is known, the evolution of the dynamic system does not depend on the history of states, actions and observations.
- The current state contains all the information needed to predict the future.
- Future states are conditionally independent of past states and past observations given the current state.
- The knowledge about the current state makes past, present and future observations statistically independent.

Therefore, the agent is able to fully understand the current state, this doesn't mean it can predict the next state. Before the execution of an action the agent can not know the configuration of the next state, only after that. Thanks to the Markov property we can remove all other edges between states and actions and keep only them shown in Fig.3.2.

3.1.2 Goals and Rewards

In reinforcement learning, the purpose or goal of the agent is formalized in terms of a special signal, called the reward, passing from the environment to the agent. At each time step, the reward is a simple number $R_t \in \mathbb{R}$. Informally, the agent's goal is to maximize the total amount of reward it receives.
This means maximizing not immediate reward, but cumulative reward in the long run. The use of a reward signal to formalize the idea of a goal is one of the most distinctive features of reinforcement learning. Although formulating goals in terms of reward signals might at first appear limiting, in practice it has proved to be flexible and widely applicable.

The best way to see this is to consider examples of how it has been, or could be, used. For example, to make a robot learn to walk, researchers have provided reward on each time step proportional to the robot's forward motion. In making a robot learn how to escape from a maze, the reward is often 1 for every time step that passes prior to escape; this encourages the agent to escape as quickly as possible. To make a robot learn to find and collect empty soda cans for recycling, one might give it a reward of zero most of the time, and then a reward of +1 for each can collected. One might also want to give the robot negative rewards when it bumps into things or when somebody yells at it. For an agent to learn to play checkers or chess, the natural rewards are +1 for winning, 1 for losing, and 0 for drawing and for all nonterminal positions.

The reward signal is your way of communicating to the robot what you want it to achieve, not how you want it achieved.

3.1.3 Policies and Value Functions

Almost all reinforcement learning algorithms involve estimating value functions, functions of states (or of state–action pairs) that estimate how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state). The notion of "how good" here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected return. Of course the rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to particular ways of acting, called policies.

Formally, a policy is a mapping from states to probabilities of selecting each possible action. If the agent is following policy π at time t, then $\pi(a|x)$ is the probability that $A_t = a$ if $X_t = x$. Like p, π is an ordinary function; the "|" in the middle of $\pi(a|s)$ merely reminds that it defines a probability distribution over $a \in A(x)$ for each $x \in X$. Reinforcement learning methods specify how the agent's policy is changed as a result of its experience.

The value function of a state x under a policy π, denoted $v_\pi(x)$, is the expected return when starting in x and following π thereafter. For MDPs, we can define v_π formally by:

$$v_\pi(s) = \mathbb{E}[G_t|X_t = x] = \mathbb{E}_\pi[\textstyle\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|X_t = x], \text{ for all } x \in X,$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable given that the agent follows policy π, t is any time step, G_t is the return (defined as some specific function of the reward sequence) following time t and γ is a parameter, $0 \leq \gamma \leq 1$, called the discount rate. Note that the value of the terminal state, if any, is always zero. We call the function v_π the state-value function for policy π.

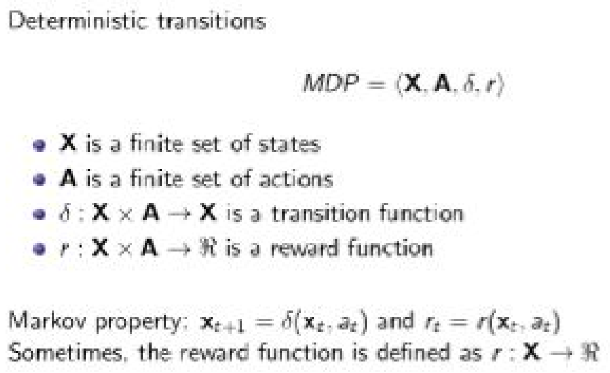
Similarly, we define the value of taking action a in state x under a policy π, denoted $q_\pi(x, a)$, as the expected return starting from x, taking the action a, and thereafter following policy π:

$$q_\pi(s, a) = \mathbb{E}[G_t|X_t = x, A_t = a] = \mathbb{E}_\pi[\textstyle\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|X_t = x, A_t = a].$$

We call q_π the action-value function for policy π.

3.1.4 Deterministic/Non-Deterministic MDP

The difference between deterministic and non-deterministic MDPs is essentially in the nature of the reward and transition functions. If only one of them is non deterministic then the MDP is non-deterministic, although in most applications it is the transition function that decides whether an MDP is deterministic or not. For this reason we will describe MDP in case of deterministic, non-deterministic and stochastic transitions.

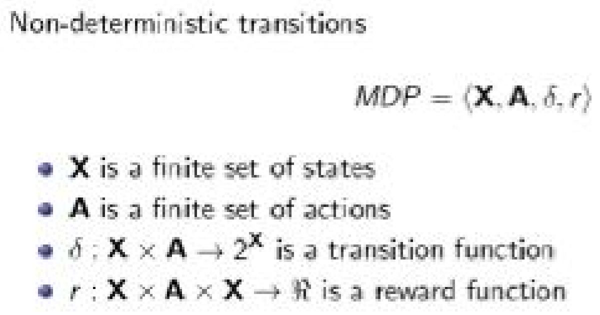

Deterministic transitions

$$MDP = \langle \mathbf{X}, \mathbf{A}, \delta, r \rangle$$

- $\mathbf{X}$ is a finite set of states
- $\mathbf{A}$ is a finite set of actions
- $\delta : \mathbf{X} \times \mathbf{A} \rightarrow \mathbf{X}$ is a transition function
- $r : \mathbf{X} \times \mathbf{A} \rightarrow \Re$ is a reward function

Markov property: $\mathbf{x}_{t+1} = \delta(\mathbf{x}_t, a_t)$ and $r_t = r(\mathbf{x}_t, a_t)$
Sometimes, the reward function is defined as $r : \mathbf{X} \rightarrow \Re$

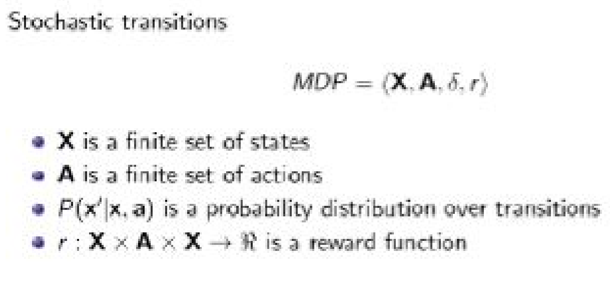
Therefore, deterministic transitions mean that if the agent decides to execute an action a_t in state x_t, it will always end up into the state x_{t+1}


Non-deterministic transitions

$$MDP = \langle \mathbf{X}, \mathbf{A}, \delta, r \rangle$$

- $\mathbf{X}$ is a finite set of states
- $\mathbf{A}$ is a finite set of actions
- $\delta : \mathbf{X} \times \mathbf{A} \rightarrow 2^{\mathbf{X}}$ is a transition function
- $r : \mathbf{X} \times \mathbf{A} \times \mathbf{X} \rightarrow \Re$ is a reward function

With non-deterministic transitions is not possible to guarantee the outcome if the agent executes an action a_t in state x_t. This is why the transition function returns a set of possible states as a result of the execution of an action.


Stochastic transitions

$$MDP = \langle \mathbf{X}, \mathbf{A}, \delta, r \rangle$$

- $\mathbf{X}$ is a finite set of states
- $\mathbf{A}$ is a finite set of actions
- $P(\mathbf{x}'|\mathbf{x}, \mathbf{a})$ is a probability distribution over transitions
- $r : \mathbf{X} \times \mathbf{A} \times \mathbf{X} \rightarrow \Re$ is a reward function

Given a state x_t and an action a_t, stochastic transitions mean what is the probability to reach a state x_{t+1}, thus it also returns a set of possible states but each one with a probability associated.

3.2 Learning with MDP

3.2.1 Dynamic Programming and Monte Carlo prediction

The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP). Classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense, but they are still important theoretically. In fact, all of these methods can be viewed as attempts to achieve much the same effect as DP, only with less computation and without assuming a perfect model of the environment. We usually assume that the environment is a finite MDP. That is, we assume that its state, action, and reward sets S, A, and R, are finite, and that its dynamics are given by a set of probabilities $p(x_0, r|x, a)$, for all $x \in X$, $a \in A(x)$, $r \in R$, and $s_0 \in S+$ ($S+$ is S plus a terminal state if the problem is episodic).

Although DP ideas can be applied to problems with continuous state and action spaces, exact solutions are possible only in special cases. A common way of obtaining approximate solutions for tasks with continuous states and actions is to quantize the state and action spaces and then apply finite-state DP methods. The goal of DP, and of reinforcement learning generally, is to make the agent follow an optimal policy in order to maximize collected rewards. To be able to achieve this goal we should solve one of the following equations:

$$v_*(x) = max\,\mathbb{E}[R_{t+1}+\gamma v_*(X_{t+1}|S_t = s, A_t = a] = \max_a \textstyle\sum_{x',r} p(x', r|x, a)[r+\gamma v_*(x')]$$

or

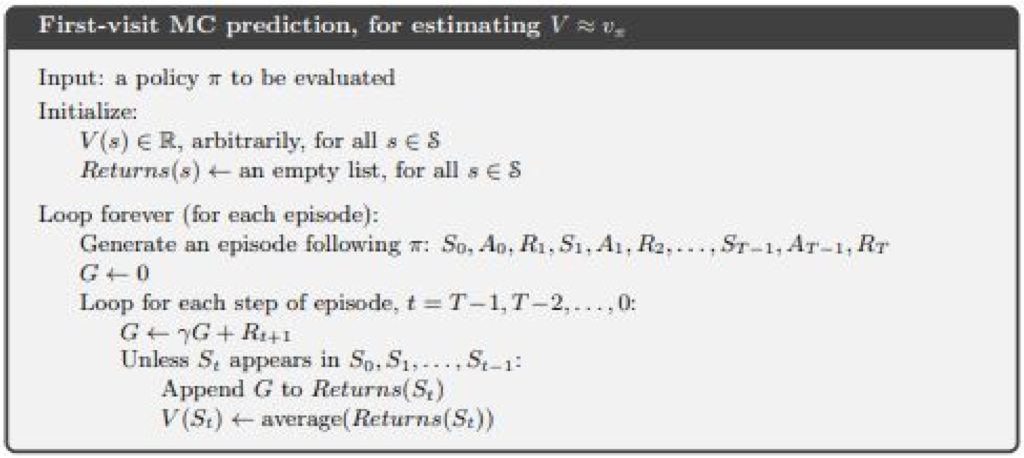
$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a')|X_t = x, A_t = a] =$$
$$\textstyle\sum_{x',r} p(x', r|x, a)[r + \gamma \max_{a'} q_*(s', a')].$$

The idea is to know the maximum value that we can get at each state. This will allow us to determine what policy to follow. As we said above, to be able to apply DP to solve the reinforcement learning problem, we need to know the transition probability matrix, as well as the reward system. We will proceed using iterative solution: we start with an initial random value at each state, then we pick a random policy to follow. The reason to have a policy is simply because in order to compute any state-value function we need to know how the agent is behaving.

Therefore we start by randomly initializing the values of all states and we call the resulting state-value function $v_0(x)$, (we call it a function because after assigning the values to all states, the $v_0(x)$ will return the value at any state x). Now we follow the policy π and on each iteration we update the values of all the states. After updating all the state, we obtain a new state-value function. The values of the states after each iteration will be closer to the theoretical values given by $v_\pi(x)$. Thus, as the iteration goes on, we will get the functions $v_1(x), v_2(x), v_3(x), \ldots.v_k(x)$, we keep iterating until the absolute difference $|v_{k-1}(x) - v_k(x)|$ for any state x, is less than a number θ. This θ is a number below which we consider that the function $v_k(x)$ has converged enough towards the $v_\pi(x)$.

All of the update estimates of the values of states based on estimates of the values of successor states. That is, they update estimates on the basis of other estimates. We call this general idea bootstrapping.
Recalling that the value of a state is the expected return expected cumulative future discounted reward starting from that state. An obvious way to estimate it from experience, then, is simply to average the returns observed after visits to that state. As more returns are observed, the average should converge to the expected value. This idea underlies all Monte Carlo methods.

In particular, suppose we wish to estimate $v_\pi(x)$, the value of a state x under policy π, given a set of episodes obtained by following π and passing through x. Each occurrence of state x in an episode is called a visit to x. Of course, x may be visited multiple times in the same episode; let us call the first time it is visited in an episode the first visit to x. The first-visit MC method estimates $v_\pi(x)$ as the average of the returns following first visits to x, whereas the every-visit MC method averages the returns following all visits to x. First-visit MC is shown in procedural form in Fig. 3.3. Every-visit MC would be the same except without the check for X_t having occurred earlier in the episode.



First-visit MC prediction, for estimating $V \approx v_\pi$

```
Input: a policy $\pi$ to be evaluated
Initialize:
    $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
    Generate an episode following $\pi$: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless $S_t$ appears in $S_0, S_1, \ldots, S_{t-1}$:
            Append $G$ to $Returns(S_t)$
            $V(S_t) \leftarrow$ average($Returns(S_t)$)
```

Figure 3.3

Both first-visit MC and every-visit MC converge to $v_\pi(x)$ as the number of visits (or first visits) to x goes to infinity. This is easy to see for the case of first-visit MC. In this case each return is an independent, identically distributed estimate of $v_\pi(x)$ with finite variance. By the law of large numbers the sequence of averages of these estimates converges to their expected value. Each average is itself an unbiased estimate, and the standard deviation of its error falls as $1/\sqrt{n}$, where n is the number of returns averaged. Every-visit MC is less straightforward, but its estimates also converge quadratically to $v_\pi(x)$ [32].

An important fact about Monte Carlo methods is that the estimates for each state are independent. The estimate for one state does not build upon the estimate of any other state, as is the case in DP. In other words, Monte Carlo methods do not bootstrap. In particular, note that the computational expense of estimating the value of a single state is independent of the number of states. This can make Monte Carlo methods particularly attractive when one requires the value of only one or a subset of states. One can generate many sample episodes starting from the states of interest, averaging returns from only these states, ignoring all others. This is a third advantage Monte Carlo methods can have over DP methods (after the ability to learn from actual experience and from simulated experience).

3.2.2 Temporal-Difference Learning

If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal difference (TD) learning. TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap).

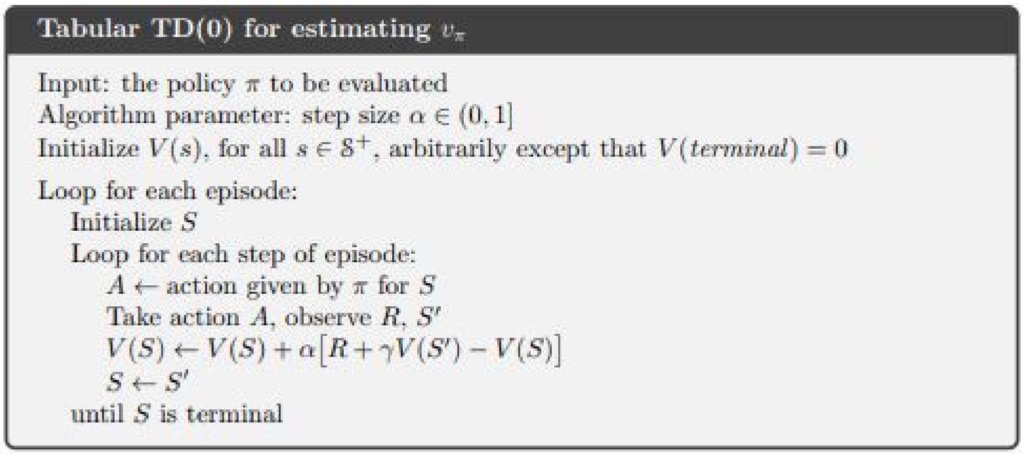
Both TD and Monte Carlo methods use experience to solve the prediction problem. Given some experience following a policy , both methods update their estimate V of v_π for the non-terminal states X_t occurring in that experience. Roughly speaking, Monte Carlo methods wait until the return following the visit is known, then use that return as a target for $V(X_t)$. A simple every-visit Monte Carlo method suitable for nonstationary environments is:

$$V(X_t) = V(X_t) + \alpha[G_t - V(X_t)]$$

Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to $V(S_t)$ (only then is G_t known), TD methods need to wait only until the next time step. At time $t+1$ they immediately form a target and make a useful update using the observed reward R_{t+1} and the estimate $V(X_{t+1})$. The simplest TD method makes the update:

$$V(X_t) = V(X_t) + \alpha[R_{t+1} + \gamma V(X_{t+1}) - V(X_t)]$$

immediately on transition to X_{t+1} and receiving R_{t+1}. In effect, the target for the Monte Carlo update is G_t, whereas the target for the TD update is $R_{t+1} + V(S_{t+1})$.



Tabular TD(0) for estimating v_π

```
Input: the policy $\pi$ to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize $S$
  Loop for each step of episode:
    $A \leftarrow$ action given by $\pi$ for $S$
    Take action $A$, observe $R$, $S'$
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until $S$ is terminal
```

Figure 3.4

This TD method is called TD(0), or one-step TD, because it is a special case of the TD(λ) and n-step TD methods. Fig. 3.4 specifies TD(0) completely in procedural form. Because TD(0) bases its update in part on an existing estimate, we say that it is a bootstrapping method, like Dynamic Programming. We know that:

$$\begin{aligned} v_\pi(x) &= \mathbb{E}_\pi[G_t|X_t = x] && (3.1)\\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|X_t = x] && (3.2)\\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(X_{t+1})|X_t = x] && (3.3) \end{aligned}$$

Roughly speaking, Monte Carlo methods use an estimate of (3.1) as a target, whereas DP methods use an estimate of (3.3) as a target. The Monte Carlo target is an estimate because the expected value in (3.1) is not known; a sample return is used in place of the real expected return. The DP target is an estimate not because of the expected values, which are assumed to be completely provided by a model of the environment, but because $v_\pi(X_{t+1})$ is not known and the current estimate, $V(X_{t+1})$, is used instead.

The TD target is an estimate for both reasons: it samples the expected values in (6.4) and it uses the current estimate V instead of the true v_π. Thus, TD methods combine the sampling of Monte Carlo with the bootstrapping of DP. Shown to the right is the backup diagram for tabular TD(0). The value estimate for the state node at the top of the backup diagram is updated on the basis of the one sample transition from it to the immediately following state. We refer to TD and Monte Carlo updates as sample updates because they involve looking ahead to a sample successor state (or state–action pair), using the value of the successor and the reward along the way to compute a backed-up value, and then updating the value of the original state (or state–action pair) accordingly.

Sample updates differ from the expected updates of DP methods in that they are based on a single sample successor rather than on a complete distribution of all possible successors. Finally, note that the quantity in brackets in the TD(0) update is a sort of error, measuring the difference between the estimated value of X_t and the better estimate $R_{t+1} + \gamma V(X_{t+1})$. This quantity, called the TD error, arises in various forms throughout reinforcement learning:

$$\delta = R_{t+1} + \gamma V(X_{t+1}) - V(X_t)$$

Notice that the TD error at each time is the error in the estimate made at that time. Because the TD error depends on the next state and next reward, it is not actually available until one time step later. That is, t is the error in $V(X_t$, available at time $t + 1$. Also note that if the array V does not change during the episode (as it does not in Monte Carlo methods), then the Monte Carlo error can be written as a sum of TD errors:

$$
\begin{aligned}
G_t - V(X_t) &= R_{t+1} + \gamma G_{t+1} - V(X_t) + \gamma V(X_{t+1}) - \gamma V(X_{t+1}) && (3.4)\\
&= \delta_t + \gamma(G_{t+1} - V(X_{t+1})) && (3.5)\\
&= \delta_t + \gamma\delta_{t+1} + \gamma^2(G_{t+2} - V(X_{t+2}) && (3.6)\\
&= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + ... + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}(G_T - V(X_T)) && (3.7)\\
&= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + ... + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}(0 - 0) && (3.8)\\
&= \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k. && (3.9)
\end{aligned}
$$

This identity is not exact if V is updated during the episode (as it is in TD(0)), but if the step size is small then it may still hold approximately. Generalizations of this identity play an important role in the theory and algorithms of temporal-difference learning.

3.2.3 Sarsa: On-policy TD Control

As with Monte Carlo methods, we face the need to trade off exploration and exploitation, and again approaches fall into two main classes: on-policy and off-policy. In this section we present an on-policy TD control method.
The first step is to learn an action-value function rather than a state-value function. In particular, for an on-policy method we must estimate $q_\pi(s, a)$ for the current behavior policy π and for all states x and actions a. In the previous section we considered transitions from state to state and learned the values of states, now we consider transitions from state–action pair to state–action pair, and learn the values of state–action pairs. Formally these cases are identical: they are both Markov chains with a reward process. The theorems assuring the convergence of state values under TD(0) also apply to the corresponding algorithm for action values:

$$Q(X_t, A_t) = Q(X_t, A_t) + \alpha[R_{t+1} + \gamma Q(X_{t+1}, A_{t+1}) - Q(X_t, A_t)].$$

This update is done after every transition from a non-terminal state X_t. If X_{t+1} is terminal, then $Q(X_{t+1}, A_{t+1})$ is defined as zero. This rule uses every element of the quintuple of events, $(X_t, A_t, R_{t+1}, X_{t+1}, A_{t+1})$, that make up a transition from one state–action pair to the next. This quintuple gives rise to the name Sarsa for the algorithm.
It is straightforward to design an on-policy control algorithm based on the Sarsa prediction method. As in all on-policy methods, we continually estimate q_π for the behavior policy π, and at the same time change π toward greediness with respect to q_π. The general form of the Sarsa control algorithm is given in the box on the next page. The convergence properties of the Sarsa algorithm depend on the nature of the policy's dependence on Q. For example, one could use ϵ-greedy or ϵ-soft policies. Sarsa converges with probability 1 to an optimal policy and action-value function as long as all state–action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (which can be arranged, for example, with ϵ-greedy policies by setting ϵ= 1/t).

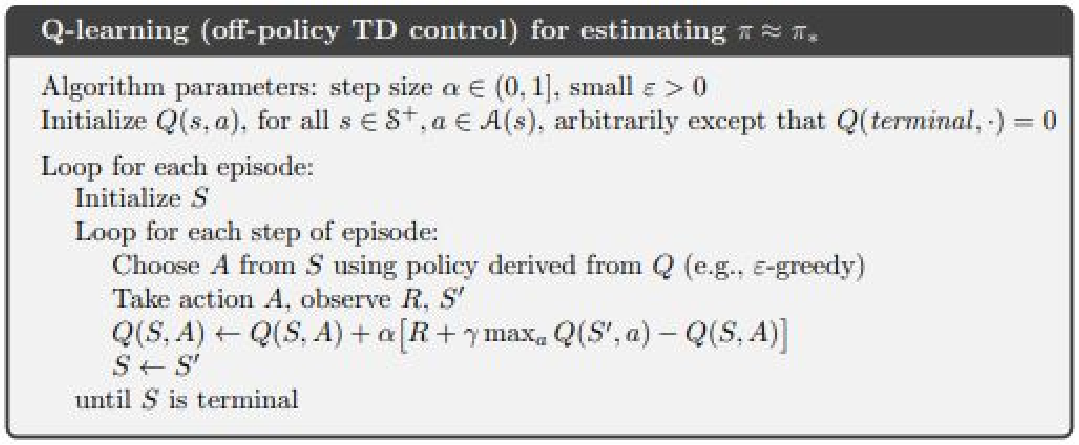
3.2.4 Q-learning: Off-policy TD Control

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning [37], defined by

$$Q(X_t, A_t) = Q(X_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(X_{t+1}, a) - Q(X_t, A_t)].$$

In this case, the learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs.

The policy still has an effect in that it determines which state–action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated. This is a minimal requirement in the sense that any method guaranteed to find optimal behavior in the general case must require it. Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, Q has been shown to converge with probability 1 to q_*. The Q-learning algorithm is shown in Fig. 5.2 in procedural form:



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

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Figure 3.5

3.2.5 Deep Reinforcement Learning

In this and next sections until the end of the chapter we will talk about the intersection of Deep Learning and Reinforcement Learning with reference mainly to [20][28]. In our reinforcement learning part of the project we did not actually implement algorithms based on deep reinforcement learning, however, to get to the solution we thought of and which we will discuss in the next chapter, we were inspired by some key mechanisms of neural networks applied to reinforcement learning.

In deep reinforcement learning neural networks are the agent that learns to map state-action pairs to rewards. Like all neural networks, they use coefficients to approximate the function relating inputs to outputs, and their learning consists to finding the right coefficients or weights by iteratively adjusting those weights along gradients that promise less error.

For example in reinforcement learning, convolutional neural networks can be used to recognize an agent's state; e.g. the screen that Super Mario is on, or the terrain before a drone. That is, they perform their typical task of image recognition. But convolutional networks derive different interpretations from images in reinforcement learning than in supervised learning. In supervised learning, the network applies a label to an image, that is, it matches names to pixels.

In fact, it will rank the labels that best fit the image in terms of their probabilities. Shown an image of donkey, it might decide the picture is 80% likely to be a donkey, 50% likely to be a horse, and 30% likely to be a dog. In reinforcement learning, given an image that represents a state, a convolutional network can rank the actions possible to perform in that state, for example, it might predict that running right will return 5 points, jumping 7 and running left none.

More specifically, in Deep Reinforcement Learning, a loss function $\mathcal{L}(y)$ measures goodness of output y, e.g. Mean Squared Error where $\mathcal{L}(y) = ||y^* - y||^2$.

$$\begin{tikzcd} x \arrow[r, "w_1"'] & h_1 \arrow[r, "w_2"'] & \dots \arrow[r, "w_n"'] & h_n \arrow[r, "W_{n+1}"'] & y \arrow[r] & \mathcal{L}(y) \end{tikzcd}$$

Figure 3.6

We want to minimize the expected loss function $\mathcal{L}(w) = \mathbb{E}_x[\mathcal{L}(y)]$ following the gradient:

$$\frac{\partial \mathcal{L}(w)}{\partial w} = \mathbb{E}_x = \begin{pmatrix} \frac{\partial \mathcal{L}(y)}{\partial w^1} \\ \vdots \\ \frac{\partial \mathcal{L}(y)}{\partial w^k} \end{pmatrix} \tag{3.10}$$

Policy π is a behavior function selecting actions given states $a = \pi(x)$.
Value function $Q^\pi(s,a)$ is expected total reward from state x and action a under policy π:

$$Q^\pi(s,a) = \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...|x,a]$$

To evaluate how good action a is in state x following policy π we use deep network that represents value function/policy/model.
Therefore, we use a deep neural network as a function approximator of the value function:

- Policy Iteration with non-linear SARSA
  - $Q(s,a,w) \approx Q^\pi(s,a)$ thus, we use the q-network on the left side to estimate the true q-values.

  - Define objective function by Mean-Squared Error in Q-values: $\mathcal{L}(w) = \mathbb{E}[(r + \gamma Q(s', a', w) - Q(s, a, w))^2]$ where $r + \gamma Q(s', a', w) - Q(s, a, w))$ is the TD-error.
  - Take the gradient $\frac{\partial \mathcal{L}(w)}{\partial w}$ and update w.

- Value iteration with non-linear Q-learning

$$\mathcal{L}(w) = \mathbb{E}[(r + \gamma max_{a'} Q(s', a', w) - Q(s, a, w))^2]$$

  where $r + \gamma max_{a'} Q(s', a', w)$ is the target. After that, compute the gradient $\frac{\partial \mathcal{L}(w)}{\partial w}$ and update the w-vector using Stochastic Gradient Descent.

Considering only this configuration of the model, Deep Reinforcement Learning can have stability issues, in fact naive Q-learning oscillates or diverges with neural networks, this is due to

- Data is sequential: successive samples are correlated (for example if a robot takes a step, and after that, it takes a second step then this latter is correlated to the former) and non-idd, this means the model may be overfitted for some class (or groups) of samples at different time and the solution will not be generalized.

- Policy changes rapidly with slight changes to Q-value, that is, policy may oscillate and the distribution of data can swing from one extreme to another (for instance if I go from left to right).

- Scale or rewards and Q-values is unknown: Naive Q-learning gradients can be large unstable when backpropagated.

For this reason, Deep Q-networks have been developed, they provide a stable solution to deep value-based reinforcement learning through:

- Use Experience Replay: it breaks the correlations in data, bring us back to idd setting (indipendent and identically distributed means samples are randomized among batches and therefore each batch has the same (or similar) data distribution, furthermore, samples are independent of each other in the same batch).

- Freeze target Q-network avoiding oscillations and breaking the correlations between Q-network and target.

- Clip rewards or normalize network adaptively to sensible range getting robust gradients.

To remove correlations, it is worth building dataset from agent's own experience. In order to follow this idea, we have to:

- take an action a_t according to ϵ-greedy policy.

- store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory D.

- sample random mini-batch of transitions (s, a, r, s') from D.
- optimise MSE between Q-network and Q-learning targets, e.g.:

$$L(w) = \mathbb{E}_{s,a,r,s'\sim D}[(r + \gamma max_{a'} Q(s', a', w) - Q(s, a, w))^2]$$

To avoid oscillations, fix parameters used in Q-learning target:

- Compute Q-learning targets with respect to old, fixed parameters w^-

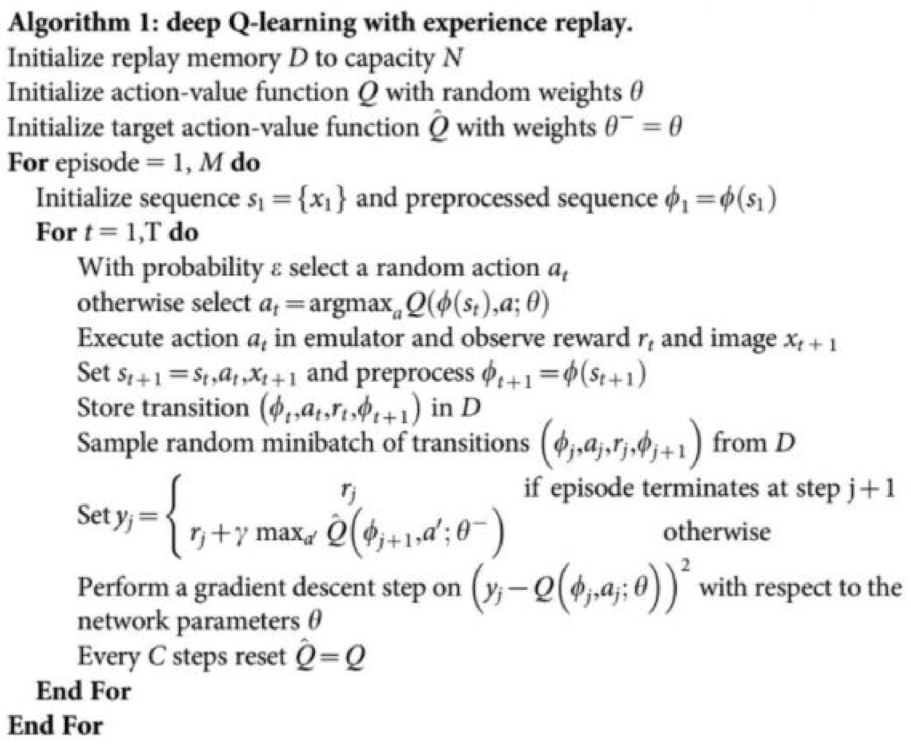
$$r + \gamma max_{a'} Q(s', a', w^-)$$

- Optimise MSE between Q-network and Q-learning targets

$$L(w) = \mathbb{E}_{s,a,r,s'\sim D}[(r + \gamma max_{a'} Q(s', a', w^-) - Q(s, a, w))^2]$$

- Periodically update fixed parameters $w^- = w$.

Therefore, Q-learning combined with experience replay and freezing target gives very impressive results. Experience replay has the largest performance improvement in DQN. Target network improvement (fixed target) is significant but not as critical as replay. However, it gets more important when the capacity of the network is small. Finally, DQN clips the reward to [-1, 1] preventing Q-values from becoming too large and ensuring gradients are well-conditioned. However, this approach can not tell the difference between small and large rewards. A better approach is normalizing network output e.g. via batch normalization.



```
**Algorithm 1: 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**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  **For** $t$ = 1,T **do**
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    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 $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
    network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  **End For**
**End For**
```

Figure 3.7

We will briefly explain how the deep-Q-learning with experience replay algorithm works referring to Fig. 3.7. The first phase is to collect data in such a way it is possible to train our q-network using memory replay. Data collection works as follow: suppose we run 10 episodes following our policy (ϵ-greedy). At the beginning we will act "pretty randomly" since we do not know anything. Thus, we pick an action randomly with probability ϵ and we will act greedy with probability $1 - \epsilon$ (in this case greedy means we use our neural network to predict the action with the highest Q-value with respect to the given input state, therefore, we are only computing the forward step giving a state in input and the network returns a number of q-values equal to the number of actions of our environment). Hence, in this first phase we want to collect data, filling a memory buffer, then we execute an action from each state getting:

$$\delta^{buffer} = [[s_1, a, r, s'], ...]$$

After collecting data, we pick randomly N samples (for instance 32) from δ^{buffer} so that samples are iid. Now we consider $y_{TARGET} = r + \gamma max_{a'} Q(s', a')$. To compute that we simply execute the forward step in the network putting s' as input and taking the highest q-value between the output values. While $y_{PREDICTED} = Q(s, a)$ that is what out network predicted always using the forward step. Thus, we want to minimize the MSE error $(y_{TARGET} - y_{PREDICTED})^2$. Having y_{TARGET} and $y_{PREDICTED}$ we can update the network (of course, we compute the error not with respect to only one sample but with respect to N samples we put in input into the network). After computing the error, we backpropagate them updating the network and we start again the process taking other N new samples from the memory buffer and we compute the new y_{TARGET} and $y_{PREDICTED}$ to update the network and so on so forth.

3.2.6 Policy Gradient Methods: REINFORCE and Actor-Critic

Until now we worked with value-based reinforcement learning algorithms. To choose which action to take given a state, we take the action with the highest Q-value (maximum expected future reward we will get at each state). As a consequence, in value-based learning, a policy exists only because of these action-value estimates. Now, we will learn a policy-based reinforcement learning technique called Policy Gradients [30].

In policy-based methods, instead of learning a value function that tells us what is the expected sum of rewards given a state and an action, we learn directly the policy function that maps state to action (select actions without using a value function). Now, the question could be why to use policy-based methods instead of Deep Q-networks or value-based methods in general. There are several reasons:

- Convergence: the problem with value-based methods is that they can have a big oscillation while training. This is because the choice of action may change dramatically for an arbitrary small change in the estimated action values.
  On the other hand, with policy gradient, we just follow the gradient to find the best parameters. We see a smooth update for our policy at each step. Because we follow the gradient to find the best parameters, we are guaranteed to converge on a local maximum (worst case) or global maximum (best case).

- Policy gradients are more effective in high dimensional action spaces: policy gradients are more effective in high dimensional action spaces or when using continuous actions. The problem with Deep Q-learning is that their predictions assign a score (maximum expected future reward) for each possible action, at each time step, given the current state. But if we have an infinite possibility of actions, for instance with a self driving car, at each state you can have a (near) infinite choice of actions like turning the wheel at 15°, 17.2°, 19.4°, honk, etc., we will need to output a Q-value for each possible action.
  On the other hand, in policy based-methods, we just adjust the parameters directly thanks to the fact that you will start to understand what the maximum will be, rather than computing (estimating) the maximum directly at every step.

- Policy gradients can learn stochastic policies: a third advantage is that policy gradient can learn a stochastic policy, while value functions can not. This has two consequences:

  – We do not need to implement an exploration/exploitation trade-off. A stochastic policy allows our agent to explore the state space without always taking the same action. This is because it outputs a probability distribution over actions. As a consequence, it handles the exploration/-exploitation trade off without hard coding it.

  – We also get ride of the problem of perceptual aliasing. Perceptual aliasing is when we have two states that seem to be (or actually are) the same, but need different actions.

Policy gradient methods has disadvantages as well: a lot of the time, they converge on a local maximum rather than on the global optimum. Instead of Deep-Q-learning, which always tries to reach the maximum, policy gradients converge slower, step by step. They can take longer to train.

Therefore, we have our policy π that has a parameter θ. This θ outputs a probability distribution of actions.

$$\pi_\theta(a|x) = P(a|x)$$

Now, to know if our policy is good we can see it as an optimization problem. We must find the best parameter θ to maximize a score function $J(\theta)$. Thus, there are two steps to find the best policy:

- Measure the quality of a π (policy) with a policy score function J(θ).

- Use policy gradient ascent to find the best parameter θ that improves our π.

The main idea here is that $J(\theta)$ will tell us how good our π is. Policy gradient ascent will help us to find the best policy parameters to maximize the sample of good actions. Then, to measure how good our policy is, we use a function called the objective function (or Policy Score Function) that calculates the expected reward of policy. Three methods work equally well for optimizing policies. The choice depends only on the environment and the objectives you have.

- In an episodic environment, we can use the start value. Calculate the mean of the return from the first time step (G_1). This is the cumulative discounted reward for the entire episode.

  $$J_1(\theta) = \mathbb{E}_\theta[G_1 = R_1 + \gamma R_2 + \gamma^2 R_3 + ...] = \mathbb{E}_\pi(V(s_1))$$

  The idea is simple: if we always start in some state s_1, we want to know what the total reward we will get from the start state until the end is. We want to find the policy that maximizes G_1, because it will be the optimal policy.

- In a continuous environment (where tasks are continuous, that is, they continue forever, there is no terminal state. For instance, for an agent that do automated stock trading, there is no starting point and terminal state, the agent keeps running until we decide to stop him), we can use the average value, because we can not rely on a specific start state. Each state value is now weighted (because some happen more than others) by the probability of the occurrence of the respected state.

  $$J_{avgv}(\theta) = \mathbb{E}_\pi(V(x)) = \sum d(x)V(x) \text{ where } d(x) = \frac{N(x)}{\sum_{x'} N(x')}$$

  where $N(x)$ is the total number of occurrences of all states and $\sum_{x'} N(x')$ is the total number of occurrences of all states.
  Or we can use the average reward per time step. The idea here is that we want to get the most reward per time step.

  $$J_{avR}(\theta) = \mathbb{E}_\pi(r) = \sum_x d(x) \sum_a \pi_\theta(x,a) R_s^a$$

  where the first sum is the probability that we are in state x, the second is the probability that we take action a from that state under policy π_θ and the last term is the immediate reward we will get.

To maximize the score function $J(\theta)$, we need to do gradient ascent on policy parameters. Gradient ascent is the inverse of gradient descent. In gradient descent, we take the direction of the steepest decrease in the function, in gradient ascent we take the direction of the steepest increase of the function. We do gradient ascent because we are not dealing with an error function, we have a score function and since we want to maximize the score, we need gradient ascent. The idea is to find the gradient to the current policy π that updates the parameters in the direction of the greatest increase and iterate.

$$\begin{gathered}\text{Policy: } \pi_\theta \\ \text{Objective Function: } J(\theta) \\ \text{Gradient: } \nabla_\theta J(\theta) \\ \text{Update: } \theta = \theta + \alpha \nabla_\theta J(\theta)\end{gathered}$$

We want to find the best parameters θ^* that maximize the score:

$$\theta^* = argmax_\theta J(\theta) = argmax_\theta\, \mathbb{E}_{\pi\theta}[\textstyle\sum_t R(x_t, a_t)]$$

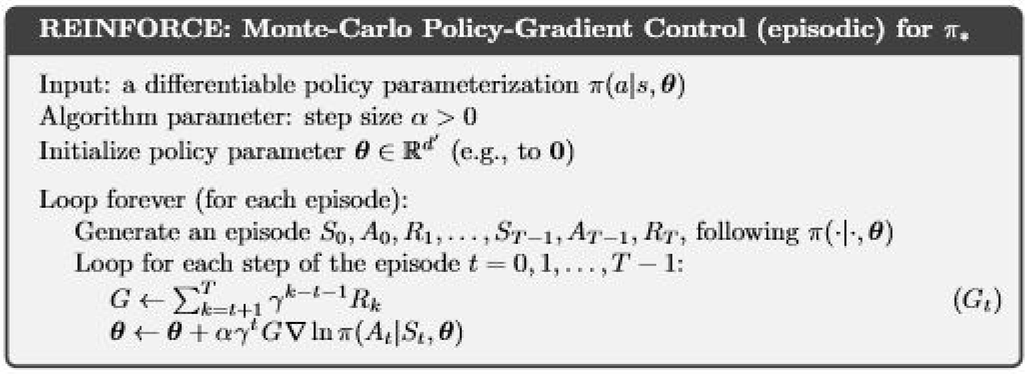
Now, using the Policy Gradient Theorem and likelihood trick, we get:

$$\text{Policy gradient: } \mathbb{E}_{\pi}[\nabla_{\theta}(log_{\pi}(x, a, \theta))R(\tau)]$$
$$\text{Update rule: } \Delta\theta = \alpha \cdot \nabla_{\theta}(log\pi(x, a, \theta))R(\tau)$$

This policy gradient is telling us how we should shift the policy distribution through changing parameters θ if we want to achieve a higher score.
$R(\tau)$ is like a scalar value score:

- if $R(\tau)$ is high, it means that on average we took actions that lead to high rewards. We want to push the probabilities of the actions seen (increase the probability of taking these actions).
- on the other hand, if $R(\tau)$ is low, we want to push down the probabilities of the actions seen.

This policy gradient causes the parameters to move most in the direction that favors actions having the highest return. Now, we describe REINFORCE, a Monte Carlo policy gradient algorithm. It simply uses Monte Carlo rollout to compute the rewards i.e. play out the whole episode to compute the total rewards.



REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

```
Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
    Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$                (G_t)
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$
```

Figure 3.8

Usually the policy π we use is soft-max. One advantage of parameterizing policies according to the soft-max in action preferences is that the approximate policy can approach a deterministic policy, whereas with ϵ-greedy action selection over action values there is always an ϵ probability of selecting a random action. In Actor-Critic model, instead of waiting until the end of the episode as we do in REINFORCE, we make an update at each step (TD learning) [29].

$$\text{Policy update: } \Delta\theta = \alpha \cdot \nabla_{\theta} \cdot (log\pi(X_t, A_t, \theta))$$
$$\text{New Update: } \Delta\theta = \alpha \cdot \nabla_{\theta} \cdot (log\pi(X_t, A_t, \theta)) \cdot Q(X_t, A_t)$$

Because we do an update at each time step, we can not use the total reward $R(t)$. Instead, we need to train a Critic model that approximates the value function (remember that the value function calculates what is the maximum expected future reward given a state and an action). This value function replaces the reward function in policy gradient that calculates the rewards only at the end of the episode.

To understand how actor critic works, imagine we play a videogame with a friend that provides us some feedback. We are the Actor and our friend is the Critic. At the beginning, we do not know how to play, thus we try some action randomly. The Critic observes our action and provides feedback. Learning from this feedback, we will update our policy and be better at playing that game. On the other hand, our friend (Critic) will also update their own way to provide feedback so it can be better next time.
As we can see, the idea of Actor Critic is to have two neural networks. We estimate both:

Actor: a policy function, it controls how our agent acts $\pi(x, a, \theta)$
Critic: a value function, it measures how good these actions are $\hat{q}(x, a, w)$

Both run in parallel. Because we have two models (Actor and Critic) that must be trained, it means that we have two set of weights (θ for our Actor and w for our Critic) that must be optimized separately:

- Policy Update:

$$\Delta\theta = \alpha\nabla_\theta(log\pi_\theta(x, a))\hat{q}_w(x, a) \tag{3.11}$$

- Value Update:

$$\Delta w = \beta(R(x, a) + \gamma\hat{q}_w(x_{t+1}, a_{t+1}) - \hat{q}_w(x_t, a_t))\nabla_w\hat{q}_w(x_t, a_t) \tag{3.12}$$

where $\hat{q}(x, a)$ is the q-learning function approximation (estimate action value), $R(x, a)+\gamma\hat{q}_w(x_{t+1}, a_{t+1})-\hat{q}_w(x_t, a_t)$ is the TD-error and $\Delta_w\hat{q}_w(x_t, a_t)$ is the gradient of our value function.
Therefore, at each time-step t, we take the current state (X_t) from the environment and pass it as an input through our Actor and our Critic. Our policy takes the state, outputs an action A_t and receives a new state (X_{t+1}) and a reward (R_{t+1}).
Thanks to that:

- the Critic computes the value of taking that action at that state.
- the Actor updates its policy parameters (weights) using the formula (3.11).

Thanks to its updated parameters, the Actor produces the next action to take at A_{t+1} given the new state S_{t+1}. The critic updates its value parameters according to formula (3.12). Hence, from the formula it is possible to notice that we are using SARSA, then we are using an on-policy approach.

3.2.7 State of the Art Algorithm: Proximal Policy Optimization

The central idea of Proximal Policy Optimization (PPO) is to avoid having too large policy update. To do that, we use a ratio that will tell us the difference between our new and old policy and clip this ratio from 0.8 to 1.2. Doing that will ensure that our policy update will not be too large.
In policy gradient methods we have discussed so far there could be a problem related to the step size: if too small, the training process is too slow, if too high, there is

too much variability in the training. That is why PP0 is useful, the idea is that PP0 improves the stability of the Actor training by limiting the policy update at each training step. To be able to do that PPO introduces a new objective function called "Clipped surrogate objective function" that will constraint the policy change in a small range using a clip.

Therefore, instead of using $log\pi$ to trace the impact of the actions, we can use the ratio between the probability of action under current policy divided by the probability of the action under previous policy.

$$r_t(\theta) = \frac{\pi_\theta(a_t|x_t)}{\pi_{\theta_{old}}(a_t|x_t)} \text{ so } r(\theta_{old}) = 1$$

- if $r_t(\theta) > 1$, it means the action is more probable in the current policy than the old policy.
- if $r_t(\theta)$ is between 0 and 1, it means that the action is less probable for current policy than for the old one.

As consequence, our new objective function could be:

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t[\frac{\pi_\theta(a_t|x_t)}{\pi_{\theta_{old}}(a_t,x_t}\hat{A}_t] = \hat{\mathbb{E}}_t[r_t(\theta)\hat{A}_t]$$

where CPI stands for conservative policy iteration.

However, without a constraint, if the action taken is much more probable in our current policy than in our former, this would lead to a large policy gradient step and consequence an excessive policy update.
Consequently, we need to constraint this objective function by penalize changes that lead to a ratio that will away from 1 (in the paper [27] the ratio can only vary from 0.8 to 1.2). By doing that we will ensure that not having too large policy update because the new policy can not be too different from the older one.
To do that we have two solutions:

- TRPO (Trust Region Policy Optimization) uses KL divergence constraints outside of the objective function to constraint the policy update. But this method is much complicated to implement and it takes more computation time.
- PPO clip probability ratio directly in the object function with its Clipped surrogate objective function:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

With Clipped Surrogate Objective Function, we have two probability ratios, one non clipped and one clipped in a range (between [1-ϵ, 1+ϵ], epsilon is an hyperparameter that helps us to define this clip range (in the paper $\epsilon = 0.2$).
Then, we take the minimum of the clipped and non clipped objective, thus the final objective is a lower bound (pessimistic bound) of the unclipped objective.

Consequently, we have two cases to consider:

- when the advantage is greater than zero: if $\hat{A}_t > 0$, it means that the action is better than the average of all the actions in that state. Therefore, we should encourage our new policy to increase the probability of taking that action at that state. Consequently, it means increasing r_t, because we increase the probability at new policy (because $A_t\cdot$ new policy) and the denominator old policy stay constant.

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t[\frac{\pi_\theta(a_t|x_t)}{\pi_{\theta_{old}}(a_t|x_t)}\hat{A}_t] = \hat{E}_t[r_t(\theta)\hat{A}_t]$$

  However, because of the clip, $r_t(\theta)$ will only grow to as much as $1+\epsilon$. It means that this action can not be 100x more probable compared to old policy (because of the clip). We do not want to update too much our policy. And that for a simple reason, remember that taking that action at that state is only one try, it does not mean that it will always lead to a super positive reward, thus we do not want to be too much greedy because it can lead to bad policy. To summarize, in the case of positive advantage, we want to increase the probability of taking that action at that step but not too much.

- when the advantage is smaller than zero: if $\hat{A}_t < 0$, the action should be discouraged because negative effect of the outcome. Consequently, it will be decreased (because action is less probable for current policy than for the old one) but because of the clip, r_t will only decrease to as little as $1-\epsilon$. Again, we do not want to make a big change in the policy by being too greedy, by completely reduce the probability of taking that action because it leads to negative advantage.

To summarize, thanks to this clipped surrogate objective, we restrict the range that the new policy can vary from the old one. Because we remove the incentive for the probability ratio to move outside of the interval. Since the clip have the effect to gradient. If the ratio is > 1 + e or < 1 - e the gradient will be equal to zero (no slope).
Thus, we see that both of these clipping regions prevent us from getting too greedy and trying to update too much at once, and updating outside of the region where this sample offers a good approximation.
The final Clipped Surrogate Objective Loss is:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](x_t)]$$

where $c1$ and $c2$ are coefficients, $L_t^{VF}(\theta)$ is the squared-error value loss $(V_\theta(x_t) - V_t^{TARG})^2$ and $S[\pi_\theta](x_t)$ add an entropy bonus to ensure sufficient exploitation.

Chapter 4

Design of the Solution

In this chapter we will illustrate how we reached our objective that we set ourselves at the beginning of this thesis, that is, finding a way for an agent to learn a behavior in order to disengage the physiotherapist during therapies with children, from a Machine Learning point of view. More in detail, we will illustrate our final model, which consists of a Supervised Learning section and a Reinforcement Learning one, as we mentioned in Chapter 1. We will then discuss the techniques and datasets used, focusing in particular on the algorithm we developed in the decision making phase.
In parallel, we have set ourselves another goal that has a fundamental role in the thesis which is to understand the representation of the state, that is, we have studied the importance of the features characterizing the state perceived by the agent in order to understand if there was the possibility of reducing the number of state variables considered while maintaining a level of learning appropriate to the achievement of the set objectives.

4.1 Definition of the MDP

Even if the final model present in Fig. 1.1 shows first the use of Supervised Learning techniques and then the use of Reinforcement Learning to elaborate the state perceived by the agent at a certain moment of time, in order to understand our implementation in its entirety, it is necessary to first describe the MDP we have designed, then explaining the state variables we have considered and the values they can assume, the actions the agent can perform in such a way as to pursue the goal, the reward obtained based on the state visited and the kind of transitions between one state and another.

As we mentioned earlier, due to the lack of privacy rights we have not been able to actually test our algorithm in real contexts with the presence of children.
For this reason we have designed a simulator able to replicate in the most logical way possible the behavior of children considering, in the recognition phase of the state through the sensors, the faces and voices of adults. In the next chapter we will show the results obtained both considering the simulator and testing the algorithm on adults in the laboratory of the University of Leeds.

Thus, we have formulated the problem under a Reinforcement Learning point of view, that is, the goal of the agent is staying as much as possible in states in which the child is emotionally well and relates correctly to the object considered. This potentially results in not slowing down the therapy and engaging the physiotherapist in calming the child.

4.1.1 State Variables

Deciding on the state variables to be analyzed is an arduous task especially when dealing with complex environments in which the dynamics can change due to several factors. After a careful analysis of the problem we therefore modelled our MDP considering each state made up of four features:

- Face Expression Recognition (FER): it indicates the emotional state of the child/person through the analysis of facial expressions. It basically can assume one of the following values “neutral”, “happy”, or “sad”.
- Speech Emotion Recognition (SER): it recognizes the emotional state of the child/person analysing audio clips. As FEC, it can assume one of the following values “neutral”, “happy”, or “sad”.
- Object state (OS): it indicates in which position the object is located. It can assume one of the following values “ground”, “close”, “far” that respectively mean the object is on the ground (it “probably” fell) or the object is close/far to the child/person.
- Environmental sound (ES): it suggests what kind of sound it recognizes in the environment. It can assume the values “television/radio on”, “people talking”, "environmental noise”, “falling object”, “sound object” that mean the television/radio is on, people are talking to each other, environmental noise (no specific source), the object considered during the therapy has fallen or the object emits sounds (e.g. talking teddy bear).

The first two state variables therefore serve to analyze the patient’s emotional state. The third state variable assumes the aforementioned values as we are assuming that in our environment, an object designated a priori is present in addition to the child to distract the child him/herself in crucial moments, i.e. in moments in which potentially the physiotherapist must interrupt the therapy to calm the child (this object in most cases will correspond to a toy).

Therefore, considering that the agent’s camera can resume the child’s face at a certain distance, the soft table on which the child rests so that the therapist can perform the therapy and the floor, the object will assume the values "close" for indicating the object is in the vicinity of the child, in particular that the child has a direct relationship with the object such as he/she is playing with us, "far" to indicate the object is on the soft table but to a distance for which the child is not directly engaged with the object itself and "ground" to indicate the object is on the floor.

Finally, the fourth and last feature analyzes the sounds present in the therapy room so as to be able to check whether, for example, the noise caused by the fall of an object can be related to the actual presence of the object on the floor, therefore with respect to the value of the third feature and/or the emotional state of the child.
Now that we have the complete overview to define a state, so that we can represent our MDP, it is necessary to define the set of states in which the agent can be. In this regard, our set of states is composed of all the possible combinations of values the state variables can take, with the exception of the combinations:

- FER = "happy" and SER = "sad" and vice versa: we do not take into consideration situations in which the patient's emotional state can be contradictory between two sensors. We therefore allow the agent to make more than one evaluation so that this situation cannot occur.
- OS = "close" and ES = "falling object", OS = "far" and ES = "falling object": the object considered can not be fallen to the ground if it is in the hands of the child or in his/her proximity (close) or is still on a soft table (far).

4.1.2 Human-Agent Interaction

In order for the agent's assistance during the therapy to be effective, it is necessary that the learner makes decisions and carries out concrete actions in order to attract the attention of the patient pursuing the goal. Therefore, once the features allowing the agent to analyze the surrounding environment have been defined, it is necessary to describe which behaviors can be taken by the agent itself. This translates, with regards to the definition of our MDP, in deciding what are the possible actions the agent can perform.

Since at the design stage we decided to implement our agent using a laptop webcam and microphone, we opted for speech actions, i.e. when the agent has understood the state in which it is, it will start talking. Human-Agent interaction is not a simple task to model and that is why it is currently the object of research not only in the artificial intelligence field, but in the psychology one as well: in a real environment the situations that can occur are many and very often each one requires a particular behavior that creates empathy between the agent and the human, that is, it should be reduced that sense of reluctance and untrustworthiness the person feels towards the agent, as described in [3][14].

Therefore, the actions the agent can take are:

- Compliment: the agent says something that gratifies the child/person.
- Calming action: the agent tries to calm the child/person with sentences like "Don't worry".
- Reproaching action: the agent emphasizes the child to calm down and not have certain behaviors (e.g. Don't be silly!)

The first action encourages the child to continue to behave consistently with the current behavior. It turns out to be an effective action, for example when the child

is emotionally "happy" and the considered object is not on the ground. Unlike the case in which it is "sad" and he/she is throwing the analyzed object, as it would incentivize the patient to engage the physiotherapist in performing tasks/actions not related to the therapy. The second action tries to stabilize the child emotionally avoiding a worse situation, so it is very important to make this decision when the child is "sad". In the case in which the child is "happy", for example, a feeling of mistrust towards the agent may arise.
Finally, the last action assumes particular importance when the child is emotionally stable and is aware of the fact that his/her behavior is engaging the physiotherapist, such as throwing an object on the floor.

4.1.3 Stochastic Environment Settings

In modeling this problem in the simulator that we will explain in more detail in the next chapter, we decided to consider a stochastic environment, more precisely we considered a stochastic transition function, i.e. there is a certain probability of ending in the next state x' considering the current x by running a certain action a (probability distribution over transitions), and a deterministic reward function, i.e. by visiting a given state x the same reward is always obtained.

Stochasticity lies in the fact that with 10% of probability it is possible to go from each state to any other, in this way our agent has the possibility to visit a larger number of states and consequently to improve the policy learned above all with respect to those states that are less visited. Moreover the concept of stochasticity perfectly models the real world and in our case the context we are considering, in fact, it could happen that during a therapy the perceived state could change for external factors that are not considered by the agent. For example, the agent may perceive that the child is behaving responsibly and does not slow down the therapy, but suddenly a loud roar during a storm could destabilize his/her behavior, causing him/her to cry.

As for the reward, since our goal is to ensure that the child does not distract the therapist from his/her task for as long as possible, we have considered:

- reward +1 when the agent is in a state in which the child is emotionally well.
- reward -1 when the agent is in a state in which the child is emotionally sad.
- reward 0 in all other states.

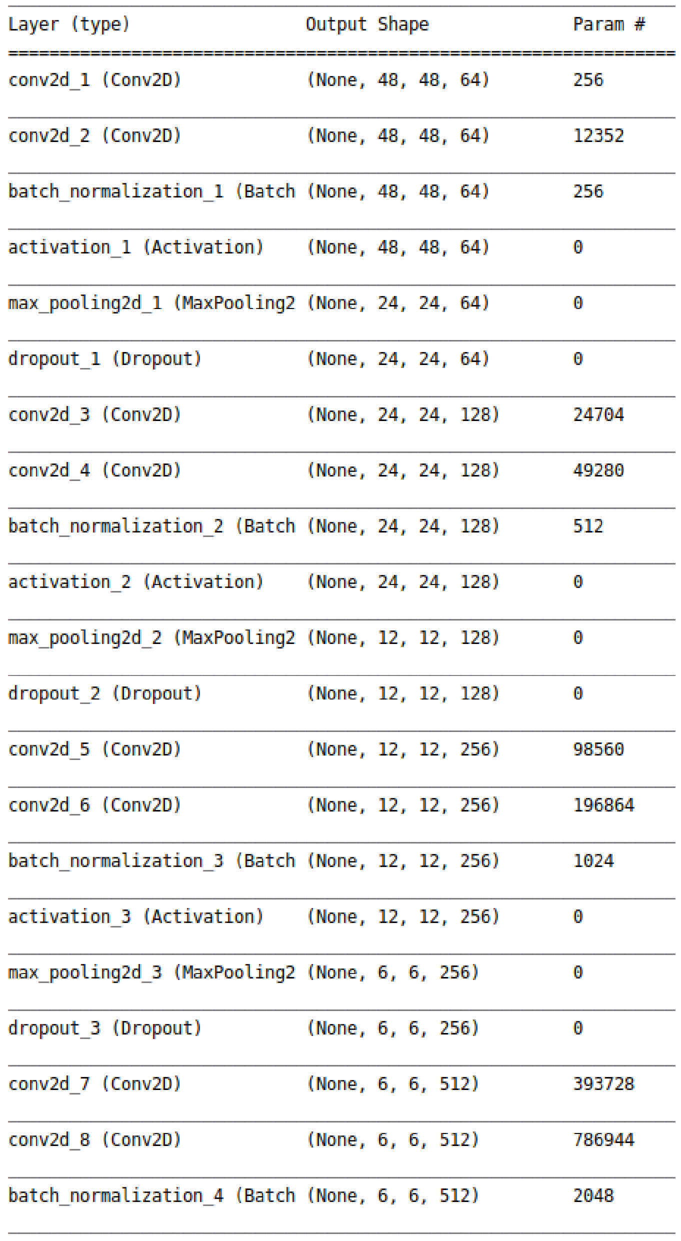
With this assignment of the reward, the agent during the learning will understand which states are more remunerative considering the reward maximization as goal, therefore, it will try to stand for as many iterations as possible in those states.

4.2 Extracting Features from Sensors

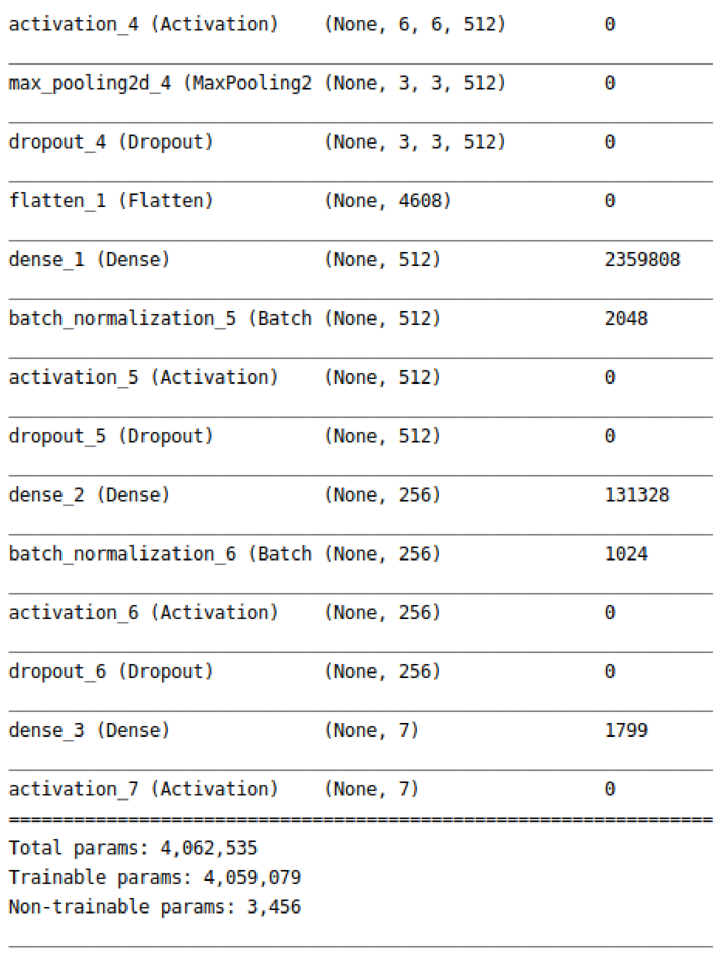
After describing our problem and explaining how we designed our MDP, we can finally describe the module concerning the part of Supervised Learning, that is how we actually get the values characterizing the state variables. We essentially used neural networks for Face Expression Recognition and Speech Emotion Recognition. We will then describe the structure of the models used and the corresponding datasets considered. Finally in the last subsection we will talk about future implementations that substantially concern the extraction of information always through sensors of the Object State and Environmental Sound features.

4.2.1 Face Expression Recognition

To implement this classification task we decided to use a pre-trained neural network presented at the following address [19] designed to solve a Kaggle competition regarding the challenges in Represention Learning [13].



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 48, 48, 64)        256
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 48, 48, 64)        12352
_________________________________________________________________
batch_normalization_1 (Batch (None, 48, 48, 64)        256
_________________________________________________________________
activation_1 (Activation)    (None, 48, 48, 64)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 24, 24, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 24, 24, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 24, 24, 128)       24704
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 24, 24, 128)       49280
_________________________________________________________________
batch_normalization_2 (Batch (None, 24, 24, 128)       512
_________________________________________________________________
activation_2 (Activation)    (None, 24, 24, 128)       0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 12, 12, 128)       0
_________________________________________________________________
dropout_2 (Dropout)          (None, 12, 12, 128)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 12, 12, 256)       98560
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 12, 12, 256)       196864
_________________________________________________________________
batch_normalization_3 (Batch (None, 12, 12, 256)       1024
_________________________________________________________________
activation_3 (Activation)    (None, 12, 12, 256)       0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 6, 6, 256)         0
_________________________________________________________________
dropout_3 (Dropout)          (None, 6, 6, 256)         0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 6, 6, 512)         393728
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 6, 6, 512)         786944
_________________________________________________________________
batch_normalization_4 (Batch (None, 6, 6, 512)         2048
_________________________________________________________________
```



```
activation_4 (Activation)    (None, 6, 6, 512)         0
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 3, 3, 512)         0
_________________________________________________________________
dropout_4 (Dropout)          (None, 3, 3, 512)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 4608)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               2359808
_________________________________________________________________
batch_normalization_5 (Batch (None, 512)               2048
_________________________________________________________________
activation_5 (Activation)    (None, 512)               0
_________________________________________________________________
dropout_5 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 256)               131328
_________________________________________________________________
batch_normalization_6 (Batch (None, 256)               1024
_________________________________________________________________
activation_6 (Activation)    (None, 256)               0
_________________________________________________________________
dropout_6 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_3 (Dense)              (None, 7)                 1799
_________________________________________________________________
activation_7 (Activation)    (None, 7)                 0
=================================================================
Total params: 4,062,535
Trainable params: 4,059,079
Non-trainable params: 3,456
_________________________________________________________________
```

Figure 4.1. FER Convolutional Network Architecture

It is a Convolutional Neural Network used as a real-time face detector (each face detected is characterized by a bounding box) and classifier of emotions. It has been programmed using Python, the real-time library for computer vision OpenCV [4] and the high-level neural networks API Keras [7]. We therefore imported the model.json and weights.h5 files into our project which, as described by the author, respectively represent the neural network architecture and the trained model weights. The network structure is presented in Fig. 4.1.

As can be seen, the network uses a large number of layers. As we know, convolutional neural networks can lead to learn a very high number of parameters, which is why we use techniques such as max-pooling as we have already described in chapter 1 in order to reduce the complexity of the input. The model is therefore composed of around four million trainable parameters.

The training of this network took place using the FER (Facial Expression Recognition) dataset always hosted in the aforementioned Kaggle competition. The data consists of 48x48 pixels grayscale images of faces. In each image faces are centered and occupy about the same amount of space.

Each image can belong to one of the following labels (0 = Angry, 1 = Disgust, 2 = Fear, 3 = Happy, 4 = Sad, 5 = Surprise, 6 = Neutral). The training set consists of 28,709 examples and the test set of 3589 examples.

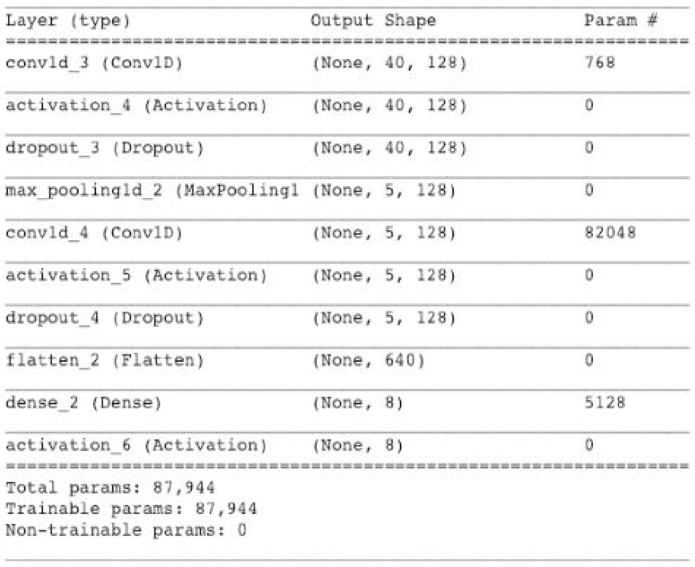
Since our first state variable can assume one of three values, while this type of neural network classifies seven different emotions, we have decided to map Angry, Disgust, Fear and Sad to the value of our feature "Sad", Happy, Surprise to our "Happy" value and Neutral to our "Neutral" value. In our final demo in which we actually use the network to classify images in real time using the camera, we obtained excellent results with regards to the classification of Neutral and Happy emotions.

Instead we have got acceptable results with regards to the classification of the Sad emotion, in particular we have attributed this behavior to the difficulty in simulating a sad face, in fact having the corners of the mouth downwards or overlapping the upper lip to the lower one do not involve a strong modification of the lips position that can always differentiate from a neutral face. For example, a smile is different to express happiness where opening the mouth by modifying the pixels is quite obvious.

The emotion that instead is classified in most cases in a correct way and that as we have just said falls into the Sad category is "Angry". In this case, in fact, the eyebrows are lowered and tend to join the center, clearly changing their position with respect to a neutral expression.

4.2.2 Speech Emotion Recognition

To classify the kind of emotion experienced by patients by analyzing their way of speaking we decided to use the pre trained network present at this address [9] because, as the author himself shows through experiments, it has remarkable performances, in fact , the model has an accuracy of around 90% in classifying between different possible emotions. The model used is therefore the following:



```
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_3 (Conv1D)            (None, 40, 128)           768
_________________________________________________________________
activation_4 (Activation)    (None, 40, 128)           0
_________________________________________________________________
dropout_3 (Dropout)          (None, 40, 128)           0
_________________________________________________________________
max_pooling1d_2 (MaxPooling1 (None, 5, 128)            0
_________________________________________________________________
conv1d_4 (Conv1D)            (None, 5, 128)            82048
_________________________________________________________________
activation_5 (Activation)    (None, 5, 128)            0
_________________________________________________________________
dropout_4 (Dropout)          (None, 5, 128)            0
_________________________________________________________________
flatten_2 (Flatten)          (None, 640)               0
_________________________________________________________________
dense_2 (Dense)              (None, 8)                 5128
_________________________________________________________________
activation_6 (Activation)    (None, 8)                 0
=================================================================
Total params: 87,944
Trainable params: 87,944
Non-trainable params: 0
_________________________________________________________________
```

Unlike the neural network used for face expression recognition, this presents a lower complexity using only two convolutional layers and a total of 87.944 trainable parameters. We imported the model defined in the Emotion_Voice_Detection_Model.h5 file and made the predictions by passing as input a wav file obtained by recording with the laptop's microphone for t seconds, where t is a parameter that can be modified at will.

In our case, in particular, we used a multi-thread application capable of simultaneously recording a video via webcam for facial expression recognition and an audio for the emotional recognition of the speech by microphone every 5 seconds.
In order to design this network, Python and Keras were used in this case as well.
The dataset used to train the model is RAVDESS [16]. It contains 7356 files produced through the participation of 24 actors who played some sentences expressing different emotions. More specifically, the classes that the model considers and is able to predict are 0 = neutral, 1 = calm, 2 = happy, 3 = sad, 4 = angry, 5 = fearful, 6 = disgust, 7 = surprised). Also in this case, as for the first state variable, we mapped the neutral and calm values to the value of our feature speech emotion recognition "Neutral", happy and surprised to "Happy", sad, angry, fearful and disgust to "Sad".

After a series of tests to be able to actually design a demo in which this and the previous model on Face Expression Recognition could cooperate together, we noticed that in general we get good results for the classification. By the way, we recall that the extraction of features through sensors is not the main focus of this thesis but it is a module we implemented in order to design a complete final system that could operate in the real world.

4.2.3 Future Implementations

In this section we want to explain what are the future implementations regarding the extraction of values of the state variables considered through the use of sensors. The main idea is to develop the remaining two state variables, namely Object State and Environmental Noise.

As for the Object state feature, in order for the implementation to be effective in a real environment, it is necessary to modify the hardware and use a depth camera, i.e. a camera that allows to calculate the distance between the identified objects and the camera itself. Combining this type of sensor with an object detection network such as YOLO [23], possibly trained on a particular object of interest, we would be able to calculate the effective distance between the patient and the object. At that point, considering certain thresholds within which the object can be held by the patient or not, or if the object is actually on the floor or on a soft table, we can assign the correct value to the feature.

As for the last state variable, it is possible to use the already available microphone as hardware and train a neural network using different datasets with respect to the values we want to assign to the Environmental Sound feature.

4.3 Making Decisions

During the development of this project we initially deepened the deep q-learning and studied the most important algorithms based on neural networks that are currently the main focus in research. Since the main objective of this thesis was not only to learn a policy using a model, but also to understand if it was possible to reduce the complexity of the state in which the agent is, getting pretty much the same results of the original state representation under the aspect of achieving the goal set by us.

Therefore, in the early stage, we studied representation learning, i.e. what it means to learn a good representation. In [2], the authors carried out an experiment regarding representation learning considering an artificial environment (the four-room domain) in which they trained a very large deep network to perform a variety of predictions. They defined a representation as the mapping between the initial discrete states of the domain and d-dimensional feature vectors (where d represents the number of features we want to learn) which in turn are mapped linearly to predictions. In essence, the network learns d features and gives you back how much they are involved in minimizing the loss function. The authors state that an optimal representation should minimize the error over all "achievable" value functions, where by "achievable" they mean "generated by some policy". In particular they consider a special subset of value functions called AVF (adversarial value functions) which can be obtained by studying the space of value functions. The latter, under a geometrical aspect, is a polytope having certain characteristics explained in [8].

To make the concept clearer, representation learning can be illustrated using an example: consider a deep learning algorithm that is trying to identify geometric figures in an image, in order to match pixels to geometric shapes, the algorithm first needs to understand some basic features/representations of the data such as the number of corners. While traditional machine learning techniques often deal with mathematical well-structured datasets, deep learning models process data such as images or sounds that have not well-defined features. The central issue is to represent an optimal representation for the input data. In the context of deep learning, the quality of a representation is mostly given by how much it facilitates the learning process. In this way, since the learning of a neural network is related to a series of parameters and since by definition it is a black-box, it is hard to figure out how effectively the final representation is processed and how the value of the features actually change during learning.

At this point, we decided to study the importance of features through the process of feature selection using a model that, at each learning step, could explicitly motivate why a feature is more important than another following a well-defined metric. In this way we could discard the less important variables and actually check what the resulting performances are, if so, reducing the complexity of the considered state. For this reason, we have manually modeled the state by defining our state variables. Thanks to [40] we decided to use decision trees as a model that could pursue at the same time the two main objectives of the thesis. The paper discusses a multi-layer structure that recalls the architecture of a neural network where in each layer multiple random forests are used. Furthermore, the paper refers to supervised learning tasks where the structure of each decision tree does not change unlike our way of building a tree to adapt it to a Reinforcement Learning task that we will explain in the next section.

4.3.1 Dynamic Decision Tree: Our Learning Model

As we have already mentioned before, our algorithm bases its update during learning on a buffer as deep q-networks do. In particular, the buffer is filled with samples, i.e. tuples $(x, a, x', Q(x, a))$ where:

- x is the state in which the agent is currently located.
- a is the action performed in the state x.
- x' is the next state where the agent ends after executing the action a in the state x.
- $Q(x, a)$ is the value that indicates how good it is for an agent to execute the action a while it is in the state x.

following the ϵ-greedy policy. Since we are considering decision trees for regression we have that each internal node is a condition on a state variable and leaf nodes represent the q-values related to the considered actions, then in our case we have a vector of three values. Edges represent whether the conditions on the state variables values are satisfied or not. Since we used the library scikit-learn [21], each node has zero or two child nodes as shown in the example in Fig. 4.2, therefore, our final tree is a binary tree.

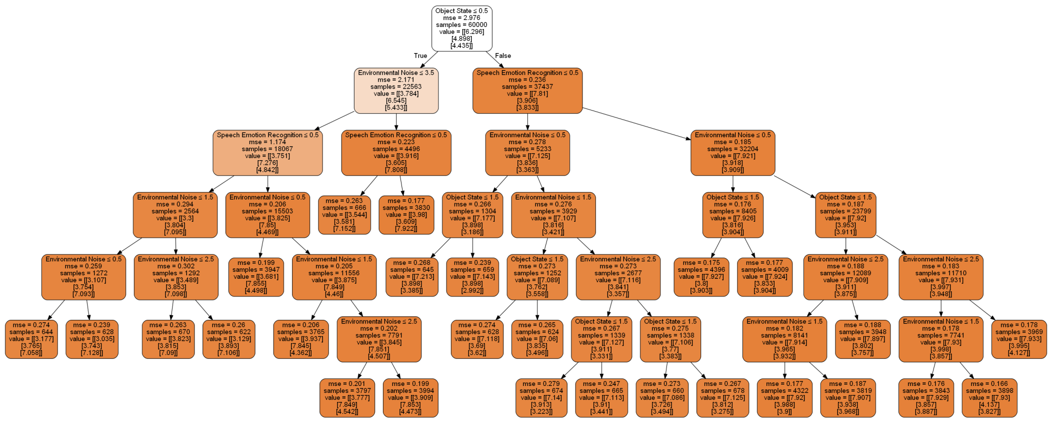

Object State ≤ 0.5 mse = 2.976 samples = 60000 value = [[6.296] [4.898] [4.435]]
True
False
Environmental Noise ≤ 3.5 mse = 2.171 samples = 22563 value = [[3.784] [6.545] [5.433]]
Speech Emotion Recognition ≤ 0.5 mse = 0.236 samples = 37437 value = [[7.81] [3.906] [3.833]]
Speech Emotion Recognition ≤ 0.5 mse = 1.174 samples = 18067 value = [[3.751] [7.276] [4.842]]
Speech Emotion Recognition ≤ 0.5 mse = 0.223 samples = 4496 value = [[3.916] [3.605] [7.808]]
Environmental Noise ≤ 0.5 mse = 0.278 samples = 5233 value = [[7.125] [3.836] [3.363]]
Environmental Noise ≤ 0.5 mse = 0.185 samples = 32204 value = [[7.921] [3.918] [3.909]]
Environmental Noise ≤ 1.5 mse = 0.294 samples = 2564 value = [[3.3] [3.804] [7.095]]
Environmental Noise ≤ 0.5 mse = 0.206 samples = 15503 value = [[3.825] [7.85] [4.469]]
mse = 0.263 samples = 666 value = [[3.544] [3.581] [7.152]]
mse = 0.177 samples = 3830 value = [[3.98] [3.609] [7.922]]
Object State ≤ 1.5 mse = 0.266 samples = 1304 value = [[7.177] [3.898] [3.186]]
Environmental Noise ≤ 1.5 mse = 0.276 samples = 3929 value = [[7.107] [3.816] [3.421]]
Object State ≤ 1.5 mse = 0.176 samples = 8405 value = [[7.926] [3.816] [3.904]]
Object State ≤ 1.5 mse = 0.187 samples = 23799 value = [[7.92] [3.953] [3.911]]
Environmental Noise ≤ 0.5 mse = 0.259 samples = 1272 value = [[3.107] [3.754] [7.093]]
Environmental Noise ≤ 2.5 mse = 0.302 samples = 1292 value = [[3.489] [3.853] [7.098]]
mse = 0.199 samples = 3947 value = [[3.681] [7.855] [4.498]]
Environmental Noise ≤ 1.5 mse = 0.205 samples = 11556 value = [[3.875] [7.849] [4.46]]
mse = 0.268 samples = 645 value = [[7.213] [3.898] [3.385]]
mse = 0.239 samples = 659 value = [[7.143] [3.898] [2.992]]
Object State ≤ 1.5 mse = 0.273 samples = 1252 value = [[7.089] [3.762] [3.558]]
Environmental Noise ≤ 2.5 mse = 0.273 samples = 2677 value = [[7.116] [3.841] [3.357]]
mse = 0.175 samples = 4396 value = [[7.927] [3.8] [3.903]]
mse = 0.177 samples = 4009 value = [[7.924] [3.833] [3.904]]
Environmental Noise ≤ 2.5 mse = 0.188 samples = 12089 value = [[7.909] [3.911] [3.875]]
Environmental Noise ≤ 2.5 mse = 0.183 samples = 11710 value = [[7.931] [3.997] [3.948]]
mse = 0.274 samples = 644 value = [[3.177] [3.765] [7.058]]
mse = 0.239 samples = 628 value = [[3.035] [3.743] [7.128]]
mse = 0.263 samples = 670 value = [[3.823] [3.815] [7.09]]
mse = 0.26 samples = 622 value = [[3.129] [3.893] [7.106]]
mse = 0.206 samples = 3765 value = [[3.937] [7.845] [4.362]]
Environmental Noise ≤ 2.5 mse = 0.202 samples = 7791 value = [[3.845] [7.851] [4.507]]
mse = 0.274 samples = 628 value = [[7.118] [3.69] [3.62]]
mse = 0.265 samples = 624 value = [[7.06] [3.835] [3.496]]
Object State ≤ 1.5 mse = 0.267 samples = 1339 value = [[7.127] [3.911] [3.331]]
Object State ≤ 1.5 mse = 0.275 samples = 1338 value = [[7.106] [3.77] [3.383]]
Environmental Noise ≤ 1.5 mse = 0.182 samples = 8141 value = [[7.914] [3.965] [3.932]]
mse = 0.188 samples = 3948 value = [[7.897] [3.802] [3.757]]
Environmental Noise ≤ 1.5 mse = 0.178 samples = 7741 value = [[7.93] [3.998] [3.857]]
mse = 0.178 samples = 3969 value = [[7.933] [3.995] [4.127]]
mse = 0.201 samples = 3797 value = [[3.777] [7.849] [4.542]]
mse = 0.199 samples = 3994 value = [[3.909] [7.853] [4.473]]
mse = 0.279 samples = 674 value = [[7.14] [3.913] [3.223]]
mse = 0.247 samples = 665 value = [[7.113] [3.91] [3.441]]
mse = 0.273 samples = 660 value = [[7.086] [3.726] [3.494]]
mse = 0.267 samples = 678 value = [[7.125] [3.812] [3.275]]
mse = 0.177 samples = 4322 value = [[7.92] [3.988] [3.9]]
mse = 0.187 samples = 3819 value = [[7.907] [3.938] [3.968]]
mse = 0.176 samples = 3843 value = [[7.929] [3.857] [3.887]]
mse = 0.166 samples = 3898 value = [[7.93] [4.137] [3.827]]

Figure 4.2. Example of binary tree returned by our algorithm

Therefore, in order to get $Q(x, a)$ it is sufficient to simply cross the tree according to the values assumed by the state variables representing x and, once the leaf node is reached, consider the value corresponding to the action a. Since there are a multitude of states, it may happen the tree divides the space of the states into regions, i.e. for a certain number of states the tree is crossed following the same path and obtaining the same value of $Q(x, a)$.

Once the buffer is filled with samples (we used a FIFO queue whose maximum size is 1000 samples), it is possible to construct a Decision Tree Regressor that approximates the Q-function and subsequently allows to generate new samples to be inserted in the buffer (We have decided that each tree generates 600 new samples), thus discarding the older ones.
Each new sample added to the buffer will have as q-value:

$$Q(x, a) = r + \gamma max_{a'} Q(x', a') \tag{4.1}$$

where:

- r is the reward obtained by reaching x' after executing the action a in the state x (we specify that the action a is chosen according to the policy considered, in our case always according to ϵ-greedy).
- γ is the discount rate, $0 \leq \gamma \leq 1$, in our case it is set to 0.9.
- $max_{a'} Q(x', a')$ is the maximum q-value among all possible actions a' executable in x'.

Once a Decision Tree Regressor has generated the new samples, it is destroyed in order to create a new decision tree considering the new buffer composed of 400 samples used by the old tree plus the 600 new ones. In order for the value function to be learned we can not build a single tree, as it would be static and would not allow us to vary the policy values. Regarding the value of epsilon, it varies over time linearly with respect to the decision trees building process.

The dynamic creation of trees occurs until the stopping criteria is satisfied, after several experiments and considerations on which type of criterion to use to stop learning we decided to consider the following one:

Every time a new tree is built, it is tested in the environment we designed considering 200 episodes and verified if the average reward is greater than 80 (after repeated tests we have confirmed that the maximum reward is around 81). If this consideration occurs for five consecutive trees then the learning following epsilon-greedy is interrupted.

Once the stopping criteria is satisfied, 100 new decision trees are built following the same criteria as above, but this time considering $\epsilon = 0$. In this way the optimal policy is reached as the exploration is null. Furthermore a new buffer is filled with all the samples generated by these trees, thus it will contain $600 \cdot 100 = 60.000$ samples and used to train a final model. Based on the type of task/experiment we want to run this new buffer has a different structure.

The experiments we will discuss in the next chapter involving the buffer structure will be:

- building trees considering all the state variables.
- building trees with lower complexity considering the feature importance.

4.3.2 Feature Importance

In the previous section we explained that after reaching the stopping criteria, our algorithm builds an additional 100 trees considering $\epsilon = 0$. The process of building those trees is not only meant to build a final buffer from which, as we will see in the next chapter, new trees can be built, but it has also the objective of studying the feature importance of the state variables considered making up the state.

Generally the importance provides a score that indicates how useful or valuable each feature is in the construction of each tree. The more an attribute is used to make key decisions in the tree, the greater its importance. This importance is explicitly calculated for each feature in the dataset considered, allowing attributes to be classified and compared to each other.

To compute the feature importance we used the feature_importances_ function implemented in the scikit-learn library. In particular, according to the scikit-learn documentation, this function returns an array of a length equal to the number of state variables considered where each value is a real number representing the percentage of importance of a feature in the corresponding tree. The sum of the array values is equal to 1. Thus the higher the value the more important is that feature. The importance $I(f)$ of a feature f is computed as the (normalized) total reduction of error brought by that feature. It is also known as the Gini importance[24]. Assuming only two child nodes (binary tree):

$$n_j = w_j c_j - w_{LEFT(j)} c_{LEFT(j)} - w_{RIGHT(j)} C_{RIGHT(j)}$$

where:

- n_j is the importance of a node j
- w_j is the weighted number of samples reaching node j
- c_j is the impurity value (MSE) of node j
- $LEFT(j)$ is the child node from left split on node j
- $RIGHT(j)$ is the child node from right split on node j

The importance for each feature on a decision tree is then computed as:

$$f_i = \frac{\sum_{\text{j:node j splits on feature i}} n_j}{\sum_{\text{j} \in \text{all nodes}} n_j}$$

where:

- f_i is the importance of feature i
- n_j is the importance of node j

These can be then normalized to a value between 0 and 1 by dividing by the sum of all feature importance values:

$$\text{norm} f_i = \frac{f_i}{\sum_{j \in \text{all features}} f_i}$$

In order to acquire a reasonable number of samples and to ascertain the actual importance of the features, we decided to sum all 100 results obtained by computing the feature importance for each tree built after satisfying the stopping criteria and setting $\epsilon = 0$. As we will explain in the next chapter, we used the Monte Carlo error as a counter-check to actually verify whether the feature importance calculation was related to learning the value function.

4.3.3 Code

In this section we explain the code portions that represent our algorithm with regard to learning an optimal policy and how the feature importance is actually computed. Since we don't initially have a tree and our buffer is empty, we can only use the exploration, setting $\epsilon = 1$ as an argument to the choose_action function. In this way we begin to add self.MAX_LEN (which corresponds to the maximum buffer size) samples, to the main buffer self.buffer by exploring the environment.

Furthermore, as we can see, two other self.X and self.y buffers are presented which contain the same number of samples as the main buffer but are structured differently. This is because the scikit-learn library allowing us to build trees requires that the input data are only the values of the features making up the state and the corresponding label, in this case the q-value. Moreover, since we initially have no q-value to update, our q-value will correspond to the reward obtained by executing the action a in the state x and ending in the next state x'.
What we have just described was implemented through the following function:

```
def initializeBuffer(self):

    i = 0
    current_state = self.env.reset()
    total_reward = 0

    while(i < self.MAX_LEN):
        action = self.choose_action(current_state, 1)
        obs, reward, done, _ = self.env.step(action)
        total_reward += reward
        q = np.zeros(self.env.action_space.n)
        q[action] = reward
        self.buffer.append([current_state, action, obs, q])
        self.X.append(current_state)
        self.y.append(q)
        current_state = obs
        i += 1
        if (done):
            current_state = self.env.reset()
```

Once the main buffer has been filled, a series of decision trees is built until the condition *avgR > TEST_REWARD* is satisfied which indicates whether the average reward obtained considering the test rewards of the last five trees is greater than 80.

```
def buildTree(self):
    self.initializeBuffer()
    self.current_tree.fit(self.X, self.y)
    i = j = lastRun = 0
    flag = False
    slidingWindowReward = []
    importance = np.zeros(len(self.features))
    while (lastRun < self.num_trees_optimal_policy):
        total_reward = 0
        epsilon = self.get_epsilon(j)
        current_state = self.env.reset()
        if (len(slidingWindowReward) == self.slidingWindow):
            avgR = sum(slidingWindowReward)/len(slidingWindowReward)
            if (avgR > TEST_REWARD):
                importance += np.array(self.current_tree.feature_importances_)
                lastRun += 1
                epsilon = 0
            else:
                slidingWindowReward = []
        while (i < self.update):
            action = self.choose_action(current_state, epsilon)
            obs, reward, done, _ = self.env.step(action)
            total_reward += reward
            q_current = self.current_tree.predict([current_state])
            q_new = self.current_tree.predict([obs])
            q_current[0][action] = reward + self.gamma * np.max(q_new)
            self.buffer.append([current_state, action, obs, q_current])
            self.X.append(current_state)
            self.y.append(q_current[0])
            if (not lastRun == 0):
                self.X_final.append(current_state)
                self.y_final.append(q_current[0])
            current_state = obs
            i += 1
            if done:
                current_state = self.env.reset()
                if(not flag):
                    flag = True
                    if (lastRun == 0):
                        slidingWindowReward.append(self.test_model)
        i = 0
        if (not flag):
            if (lastRun == 0):
                slidingWindowReward.append(total_reward)
        self.current_tree = DecisionTreeRegressor()
        self.current_tree.fit(self.X, self.y)
        j += 1
        flag = False
```

At this point, *self.num_trees_optimal_policy* trees are built by considering $\epsilon = 0$, on each of them *self.current_tree.feature_importances_* is computed and added to the values contained in the *importance* array, that is, we are summing all the feature

importances returned by the 100 trees considered.
Each tree built by the algorithm, considering the exploration or not, generates *self.update* samples to insert in the buffers *self.buffer*, *self.X* and *self.y.* Only the samples generated by the last *self.num_trees_optimal_policy* trees will be inserted in the buffers *self.X_final* and *self.y_final* subsequently used for different purposes which we will explain in the next chapter.

The main focus is the update considered in the building phase between the current tree and the next one, in fact in the innermost while the new samples, that are generated by the current tree, will have a q-value *q_current* updated by the *q_current[0][action] = reward + self.gamma * np.max (q_new)*, i.e. only the q-value corresponding to the state x and the action performed a is actually updated (leaving the values corresponding to the other actions unchanged).

4.3.4 Future Implementations

This thesis was based on developing an agent who can learn a certain behavior with respect to the context we have considered and the objective we have set ourselves. Since the goal of the project to which this thesis belongs is to implement social robots that can effectively disengage the physiotherapist during therapies with children, the first useful implementation is precisely to implement the algorithm we designed on a robot: currently the University of Leeds has a TiaGo and Sapienza a Pepper, both excellent robots for solving this kind of task.

Another interesting aspect to analyze is the study of the considered environment: in order for the agent to help the therapist as effectively as possible, a greater number of actions could be added, which in parallel with the implementation of the code on a robot, could being using the robotic arms, for example when the robot perceives the fall of the analyzed object on the floor, it could approach it and pick it up.
Finally, as we will explain later in the last Future Work chapter, there are other issues to consider, such as what actions a robot can and cannot perform in particular contexts and the kind of learning to consider to improve the solution to problems like ours.

Chapter 5

Experiments

5.1 Environment Simulator

In this section we show how we designed our environment on which we then actually trained our agent using Open AI Gym [6], a toolkit to develop reinforcement learning algorithms. Our environment essentially includes 91 states, which as we know are characterized by four state variables and 5 regions each one divided into 3 different subregions. A region is a set of states that has in common the values of the first two features, namely the emotional variables (some regions are larger since they consider more combinations of the first two variables, see for example the Neutral-Happy/Happy-Neutral region). This decision was made because we want to make sure that taking actions change the patient's mood or, at least, ensure us the agent stays in a good situation to solve our problem.
By sub-region we mean instead a set of states for which by performing an action the agent ends up in a subsequent state belonging to the same subsequent region getting the same reward, the latter depends on the type of region that is visited, more in detail, it is +1 if the agent is in a Happy-Happy region, -1 if it is in a Sad-Sad region, 0 otherwise. To better explain the concept of sub-region, consider an example: if the agent is either in the state (Neutral, Happy, Close, People talking) or (Neutral, Happy, Far, Environmental Noise) which belong to the same sub-region and it performs the Compliment action, it will then end in both cases in a state belonging to the same subsequent region which in this case is Happy-Happy obtaining the same +1 reward.

As we already mentioned in the previous chapter, the environment is stochastic, i.e. in 90% of the times in which an action is performed from a state x belonging to the region X, the agent ends up in a state y belonging to the region Y where the regions X and Y have a logical relationship linked to the problem treated, while in 10% of the times by performing an action a in state x the agent can end up in any other state of the environment. Stochasticity allows us to model those situations in which external factors not analyzed by the agent can change its perception of the environment, i.e. the value of the features observed.
In the following subsections we show the regions and states contained therein, the actions and the corresponding transitions.

5.1.1 Neutral-Neutral Region

States of Sub-Region 1			
Face Expression Recognition	Speech Emotion Recognition	Object State	Environmental Sound
Neutral	Neutral	Close	TV/Radio ON
Neutral	Neutral	Close	People talking
Neutral	Neutral	Close	Environmental Noise
Neutral	Neutral	Close	Sound Object
Neutral	Neutral	Far	TV/Radio ON
Neutral	Neutral	Far	People talking
Neutral	Neutral	Far	Environmental Noise
Neutral	Neutral	Far	Sound Object

Sub-Region 1 Transitions		
Action	Source Region	Destination Region
Compliment	Neutral-Neutral	Neutral-Happy/Happy-Neutral
Don't worry	Neutral-Neutral	Neutral-Neutral
Don't be silly	Neutral-Neutral	Neutral-Sad/Sad-Neutral

States of Sub-Region 2			
Face Expression Recognition	Speech Emotion Recognition	Object State	Environmental Sound
Neutral	Neutral	Ground	TV/Radio ON
Neutral	Neutral	Ground	People talking
Neutral	Neutral	Ground	Environmental Noise
Neutral	Neutral	Ground	Sound Object

Sub-Region 2 Transitions		
Action	Source Region	Destination Region
Compliment	Neutral-Neutral	Neutral-Sad/Sad-Neutral
Don't worry	Neutral-Neutral	Neutral-Neutral
Don't be silly	Neutral-Neutral	Neutral-Happy/Happy-Neutral

States of Sub-Region 3			
Face Expression Recognition	Speech Emotion Recognition	Object State	Environmental Sound
Neutral	Neutral	Ground	Falling Object

Sub-Region 3 Transitions		
Action	Source Region	Destination Region
Compliment	Neutral-Neutral	Neutral-Sad/Sad-Neutral
Don't worry	Neutral-Neutral	Neutral-Sad/Sad-Neutral
Don't be silly	Neutral-Neutral	Neutral-Happy/Happy-Neutral

5.1.2 Neutral-Happy/Happy-Neutral Region

States of Sub-Region 1			
Face Expression Recognition	Speech Emotion Recognition	Object State	Environmental Sound
Neutral	Happy	Close	TV/Radio ON
Neutral	Happy	Close	People talking
Neutral	Happy	Close	Environmental Noise
Neutral	Happy	Close	Sound Object
Neutral	Happy	Far	TV/Radio ON
Neutral	Happy	Far	People talking
Neutral	Happy	Far	Environmental Noise
Neutral	Happy	Far	Sound Object
Happy	Neutral	Close	TV/Radio ON
Happy	Neutral	Close	People talking
Happy	Neutral	Close	Environmental Noise
Happy	Neutral	Close	Sound Object
Happy	Neutral	Far	TV/Radio ON
Happy	Neutral	Far	People talking
Happy	Neutral	Far	Environmental Noise
Happy	Neutral	Far	Sound Object

Sub-Region 1 Transitions		
Action	Source Region	Destination Region
Compliment	Neutral-Happy/Happy-Neutral	Happy-Happy
Don't worry	Neutral-Happy/Happy-Neutral	Neutral-Happy/Happy-Neutral
Don't be silly	Neutral-Happy/Happy-Neutral	Neutral-Neutral

States of Sub-Region 2			
Face Expression Recognition	Speech Emotion Recognition	Object State	Environmental Sound
Neutral	Happy	Ground	TV/Radio ON
Neutral	Happy	Ground	People talking
Neutral	Happy	Ground	Environmental Noise
Neutral	Happy	Ground	Sound Object
Happy	Neutral	Ground	TV/Radio ON
Happy	Neutral	Ground	People talking
Happy	Neutral	Ground	Environmental Noise
Happy	Neutral	Ground	Sound Object

Sub-Region 2 Transitions		
Action	Source Region	Destination Region
Compliment	Neutral-Happy/Happy-Neutral	Neutral-Neutral
Don't worry	Neutral-Happy/Happy-Neutral	Neutral-Happy/Happy-Neutral
Don't be silly	Neutral-Happy/Happy-Neutral	Happy-Happy

States of Sub-Region 3			
Face Expression Recognition	Speech Emotion Recognition	Object State	Environmental Sound
Neutral	Happy	Ground	Falling Object
Happy	Neutral	Ground	Falling Object

Sub-Region 3 Transitions		
Action	Source Region	Destination Region
Compliment	Neutral-Happy/Happy-Neutral	Neutral-Neutral
Don't worry	Neutral-Happy/Happy-Neutral	Neutral-Neutral
Don't be silly	Neutral-Happy/Happy-Neutral	Happy-Happy

5.1.3 Happy-Happy Region

States of Sub-Region 1			
Face Expression Recognition	Speech Emotion Recognition	Object State	Environmental Sound
Happy	Happy	Close	TV/Radio ON
Happy	Happy	Close	People talking
Happy	Happy	Close	Environmental Noise
Happy	Happy	Close	Sound Object
Happy	Happy	Far	TV/Radio ON
Happy	Happy	Far	People Talking
Happy	Happy	Far	Environmental Noise
Happy	Happy	Far	Sound Object

Sub-Region 1 Transitions		
Action	Source Region	Destination Region
Compliment	Happy-Happy	Happy-Happy
Don't worry	Happy-Happy	Neutral-Happy/Happy-Neutral
Don't be silly	Happy-Happy	Neutral-Happy/Happy-Neutral

States of Sub-Region 2			
Face Expression Recognition	Speech Emotion Recognition	Object State	Environmental Sound
Happy	Happy	Ground	TV/Radio ON
Happy	Happy	Ground	People talking
Happy	Happy	Ground	Environmental Noise
Happy	Happy	Ground	Sound Object

Sub-Region 2 Transitions		
Action	Source Region	Destination Region
Compliment	Happy-Happy	Neutral-Happy/Happy-Neutral
Don't worry	Happy-Happy	Happy-Happy
Don't be silly	Happy-Happy	Neutral-Happy/Happy-Neutral

States of Sub-Region 3			
Face Expression Recognition	Speech Emotion Recognition	Object State	Environmental Sound
Happy	Happy	Ground	Falling Object

Sub-Region 3 Transitions		
Action	Source Region	Destination Region
Compliment	Happy-Happy	Neutral-Happy/Happy-Neutral
Don't worry	Happy-Happy	Neutral-Happy/Happy-Neutral
Don't be silly	Happy-Happy	Happy-Happy

5.1.4 Neutral-Sad/Sad-Neutral Region

States of Sub-Region 1			
Face Expression Recognition	Speech Emotion Recognition	Object State	Environmental Sound
Neutral	Sad	Close	TV/Radio ON
Neutral	Sad	Close	People talking
Neutral	Sad	Close	Environmental Noise
Neutral	Sad	Close	Sound Object
Neutral	Sad	Far	TV/Radio ON
Neutral	Sad	Far	People talking
Neutral	Sad	Far	Environmental Noise
Neutral	Sad	Far	Sound Object
Sad	Neutral	Close	TV/Radio ON
Sad	Neutral	Close	People talking
Sad	Neutral	Close	Environmental Noise
Sad	Neutral	Close	Sound Object
Sad	Neutral	Far	TV/Radio ON
Sad	Neutral	Far	People talking
Sad	Neutral	Far	Environmental Noise
Sad	Neutral	Far	Sound Object

Sub-Region 1 Transitions		
Action	Source Region	Destination Region
Compliment	Neutral-Sad/Sad-Neutral	Neutral-Sad/Sad-Neutral
Don't worry	Neutral-Sad/Sad-Neutral	Neutral-Neutral
Don't be silly	Neutral-Sad/Sad-Neutral	Sad-Sad

States of Sub-Region 2			
Face Expression Recognition	Speech Emotion Recognition	Object State	Environmental Sound
Neutral	Sad	Ground	TV/Radio ON
Neutral	Sad	Ground	People talking
Neutral	Sad	Ground	Environmental Noise
Neutral	Sad	Ground	Sound Object
Sad	Neutral	Ground	TV/Radio ON
Sad	Neutral	Ground	People talking
Sad	Neutral	Ground	Environmental Noise
Sad	Neutral	Ground	Sound Object

Sub-Region 2 Transitions		
Action	Source Region	Destination Region
Compliment	Neutral-Sad/Sad-Neutral	Sad-Sad
Don't worry	Neutral-Sad/Sad-Neutral	Neutral-Sad/Sad-Neutral
Don't be silly	Neutral-Sad/Sad-Neutral	Neutral-Neutral

States of Sub-Region 3			
Face Expression Recognition	Speech Emotion Recognition	Object State	Environmental Sound
Neutral	Sad	Ground	Falling Object
Sad	Neutral	Ground	Falling Object

Sub-Region 3 Transitions		
Action	Source Region	Destination Region
Compliment	Neutral-Sad/Sad-Neutral	Sad-Sad
Don't worry	Neutral-Sad/Sad-Neutral	Sad-Sad
Don't be silly	Neutral-Sad/Sad-Neutral	Neutral-Neutral

5.1.5 Sad-Sad Region

States of Sub-Region 1			
Face Expression Recognition	Speech Emotion Recognition	Object State	Environmental Sound
Sad	Sad	Close	TV/Radio ON
Sad	Sad	Close	People talking
Sad	Sad	Close	Environmental Noise
Sad	Sad	Close	Sound Object
Sad	Sad	Far	TV/Radio ON
Sad	Sad	Far	People Talking
Sad	Sad	Far	Environmental Noise
Sad	Sad	Far	Sound Object

Sub-Region 1 Transitions		
Action	Source Region	Destination Region
Compliment	Sad-Sad	Sad-Sad
Don't worry	Sad-Sad	Neutral-Sad/Sad-Neutral
Don't be silly	Sad-Sad	Sad-Sad

States of Sub-Region 2			
Face Expression Recognition	Speech Emotion Recognition	Object State	Environmental Sound
Sad	Sad	Ground	TV/Radio ON
Sad	Sad	Ground	People talking
Sad	Sad	Ground	Environmental Noise
Sad	Sad	Ground	Sound Object

Sub-Region 2 Transitions		
Action	Source Region	Destination Region
Compliment	Sad-Sad	Sad-Sad
Don't worry	Sad-Sad	Neutral-Sad/Sad-Neutral
Don't be silly	Sad-Sad	Neutral-Sad/Sad-Neutral

States of Sub-Region 3			
Face Expression Recognition	Speech Emotion Recognition	Object State	Environmental Sound
Sad	Sad	Ground	Falling Object

Sub-Region 3 Transitions		
Action	Source Region	Destination Region
Compliment	Sad-Sad	Sad-Sad
Don't worry	Sad-Sad	Sad-Sad
Don't be silly	Sad-Sad	Neutral-Sad/Sad-Neutral

5.2 Test 1: Computing the feature importance

In this first experiment we analyzed the feature importance returned by our algorithm using the average Monte Carlo error as a counter-proof. Since our environment is stochastic we decided to carry out a series of runs, more precisely, we decided to carry out 25 of them, getting the following results:

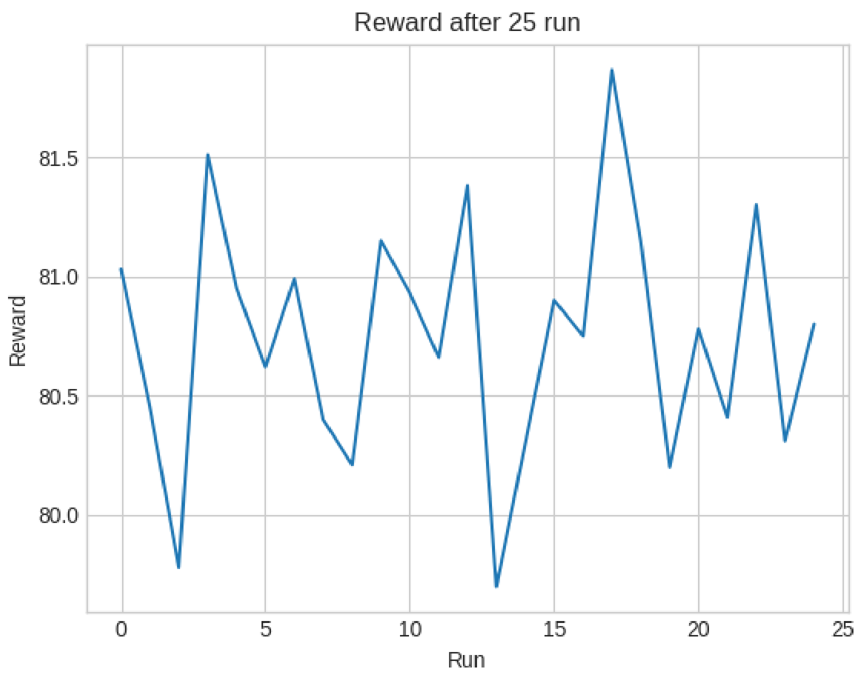


Figure 5.1. Reward considering 25 runs

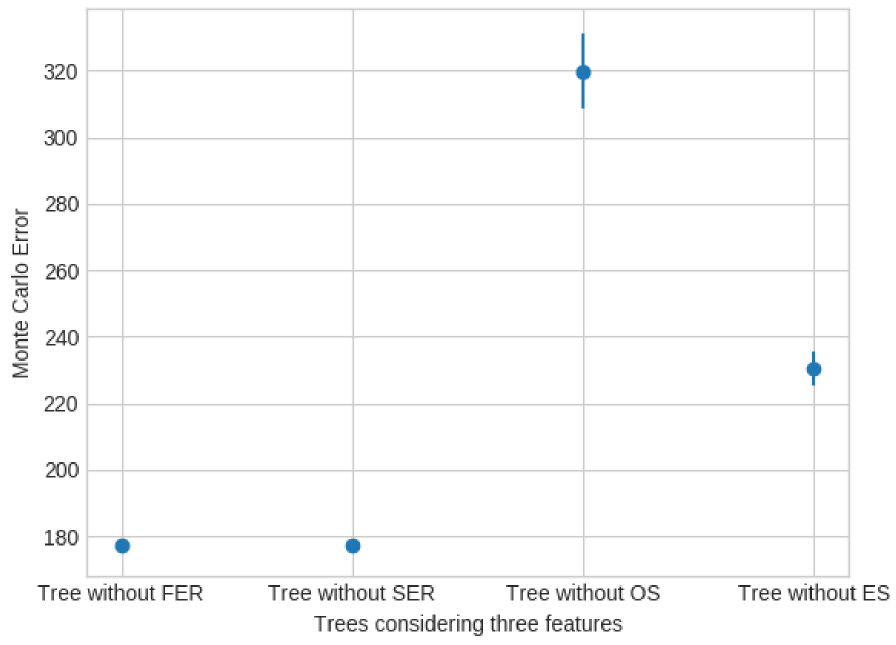


Figure 5.2. Average Monte-Carlo Error 95% confidence intervals

```
FINAL AVERAGE REWARD CONSIDERING ALL FEATURES: 80.741
TOTAL IMPORTANCE ACCORDING TO TREES:  [0.101 0.099 0.646 0.154]

Tree without considering feature: Face Expression Recognition
Average reward: 79.318
Inferior interval: 175.791 Average: 177.550 Superior Interval: 179.308

Tree without considering feature: Speech Emotion Recognition
Average reward: 79.443
Inferior interval:  175.791 Average: 177.455 Superior Interval: 179.120

Tree without considering feature: Object State
Average reward: 41.326
Inferior interval: 308.602 Average: 319.799 Superior Interval: 330.996

Tree without considering feature: Environmental Sound
Average reward: 65.776
Inferior interval: 225.403 Average: 230.392 Superior Interval: 235.382
```

As you can see from the resulting values, we have that the second feature, Speech Emotion Recognition, is the least important, in fact it has a 9% importance in the construction of the 100 trees with $\epsilon = 0$. Actually we can conclude that the first two features have the same importance and that they correspond to the two least important features. Subsequently Environmental Sound appears to have an importance of 15% and finally the most important feature is Object State with an importance of about 65%.
Computing the importance of the features is related to minimize the approximation error of the value function, in fact, a tree generated considering the most important features will have a lower average Monte Carlo error than a tree that does not. This error indicates the approximation error of the function approximator. To compute it, we considered the last buffer of 60,000 samples obtained after building the 100 trees with $\epsilon = 0$ and we built four new trees each with a missing feature. At that point we calculated the confidence intervals of the Monte Carlo error considering the formula:

$$e_t = |G_t - Q(x_t, a_t)| \qquad \text{for } t = 1, ..., 100$$
$$\text{AVG\_MC\_ERR} = \frac{1}{N}\sum_t e_t$$

where:

- N is the number of episodes.
- $G_t = r + \gamma G_{t+1}$ is the sample mean, an unbiased estimator.
- $Q(x_t, a_t)$ is a biased estimator.

Thus, this error highlights how wrong our sub-approximation is with respect to its potential to converge to the value of the policy we are considering. From the results obtained we have that actually a tree without considering feature x with a Monte Carlo error lower than a tree without considering feature y means that x is less important than y. In fact, for example, the first two trees built without considering the first two features have the same average Monte Carlo error. Obvious is the case of the tree without considering the Object State feature that has the highest possible Monte Carlo error, confirming the fact that the third feature is the most important (in fact, as already shown, it has an importance of about 65%).

With this experiment, in addition to actually verifying the functioning of our mechanism for computing the feature importance, we also asked ourselves another question: in a stochastic environment it is not sure that with a few runs an error in approximation of the value function (Monte Carlo error) entails an equal degradation of the value of the policy (reward obtained). In fact, it may happen fortuitously that a slightly worse error may correspond to a better policy value. Therefore, for this reason, we carried out the experiment considering the aforementioned number of runs in order to evaluate whether there was a relationship between the two aspects and the results confirmed that actually the decrease in the accuracy of the value function corresponds to a lower policy value.

5.3 Test 2: Training according the feature importance

In this second experiment we wonder if training considering three features immediately, instead of training trees considering all features, building 100 trees with $\epsilon = 0$ and using the final buffer of 60,000 samples considering only three features, can change the performance of the algorithm.
The following statistics show the results:

- **Tree without considering Face Expression Recognition**

```
FINAL AVERAGE REWARD CONSIDERING ALL FEATURES: 79.581
TOTAL IMPORTANCE ACCORDING TO TREES: [0.126 0.717 0.158]
MONTE CARLO ERROR CONSIDERING ALL FEATURES
Inf.Interval: 176.149 Average: 177.399 Sup.Interval: 178.648

Tree without considering feature: Speech Emotion Recognition
Average reward: 74.532
Inf.Interval: 193.742 Average: 196.597 Sup.Interval: 199.452

Tree without considering feature: Object State
Average reward: 13.814
Inf.Interval: 495.666 Average: 510.510 Sup.Interval: 525.354

Tree without considering feature: Environmental Sound
Average reward: 65.708
Inf.Interval: 200.307 Average: 214.180 Sup.Interval: 228.052
```

- **Tree without considering Speech Emotion Recognition**

```
FINAL AVERAGE REWARD CONSIDERING ALL FEATURES: 79.999
TOTAL IMPORTANCE ACCORDING TO TREES: [0.126 0.718 0.156]
MONTE CARLO ERROR CONSIDERING ALL FEATURES
Inf.Interval: 174.069 Average: 176.079 Sup.Interval: 178.089
```

```
Tree without considering feature: Face Expression Recognition
Average reward: 75.142
Inf.Interval: 195.167 Average: 197.826 Sup.Interval: 200.484

Tree without considering feature: Object State
Average reward: 14.146
Inf.Interval: 509.187 Average: 525.762 Sup.Interval: 542.338

Tree without considering feature: Environmental Sound
Average reward: 65.235
Inf.Interval: 211.832 Average: 216.257 Sup.Interval: 220.682
```

- **Tree without considering Object State**

```
FINAL AVERAGE REWARD CONSIDERING ALL FEATURES: 51.103
TOTAL IMPORTANCE ACCORDING TO TREES: [0.348 0.401 0.251]
MONTE CARLO ERROR CONSIDERING ALL FEATURES
Inf.Interval: 154.363 Average: 156.086 Sup.Interval: 157.809

Tree without considering feature: Face Expression Recognition
Average reward: 39.699
Inf.Interval: 182.117 Average: 184.897 Sup.Interval: 187.677

Tree without considering feature: Speech Emotion Recognition
Average reward: 39.428
Inf.Interval: 182.412 Average: 185.914 Sup.Interval: 189.416

Tree without considering feature: Environmental Sound
Average reward: 35.427
Inf.Interval: 178.320 Average: 182.139 Sup.Interval: 185.958
```

- **Tree without considering Environmental Sound**

```
FINAL AVERAGE REWARD CONSIDERING ALL FEATURES: 76.210
TOTAL IMPORTANCE ACCORDING TO TREES: [0.135 0.135 0.730]
MONTE CARLO ERROR CONSIDERING ALL FEATURES
Inf.Interval: 160.952 Average: 161.773 Sup.Interval: 162.595

Tree without considering feature: Face Expression Recognition
Average reward: 73.070
Inf.Interval: 169.027 Average: 170.002 Sup.Interval: 170.977

Tree without considering feature: Speech Emotion Recognition
Average reward: 72.938
Inf.Interval: 168.512 Average: 170.104 Sup.Interval: 171.696

Tree without considering feature: Object State
Average reward: 28.576
Inf.Interval: 326.453 Average: 334.161 Sup.Interval: 341.869
```

As you can see, by removing the two less important features, that is, Face Expression Recognition and Speech Emotion Recognition, we get the same value of the policy, that is the same reward of about 80 in the test phase as when training considering the original procedure. For the two most important features, however, we get a better reward by training from scratch. In fact, we get about 10 more reward points by training immediately with three features (in the tree without considering Object State we get a reward of 51.103 and in that without considering Environmental Sound we get 76.120) compared to the results of the test 1 (in the tree without Object State we get 41.326 and in the one without Environmental Sound we get 65.776). With this experiment we can affirm that the exploration has adapted during the training, therefore, it has allowed to find a policy that maximizes the use of the input features. This discussion also extends to the case of trees with only two features, from the results obtained, however, we note that in the best case, that is, considering the tree without the first two features, we get a reward of about 75, too low a value to actually reach our aim.
Since we have thus given the possibility to the policy to change as a consequence of the error of the value function, we have found a policy adapting to the fact that the value function is a little wrong and therefore the result is a policy a little better for the capacity of that value function. It can therefore be concluded that the exploration helps, but nevertheless the values of the policies obtained by removing a feature in each tree respect the feature importance computed in the first experiment, resulting in a second counter-proof in favor of the the feature importance calculation. We can therefore conclude from this experiment that to compute the feature importance it is always worth training trees with all the features, however, when we actually have the resulting most important features we must train them from scratch.

5.4 Test 3: Running a final demo

In this latest experiment we designed a demo in which the agent is trained before being actually used in a real context (offline training). Then the sensors are used to extract the values of the considered state variables classifying videos and audios representing the PhD student of the School of Computing at the University of Leeds and subsequently, based on the learned policy, the appropriate decision is made. Given the results of the feature importance in test 1, we decided to run the demo without considering the second state variable Speech Emotion Recognition due to its little importance in the learning process and to analyze five times the environment . We then trained our model from scratch according to the considerations drawn from the second experiment, getting the following behavior:

1. State = (Sad, Close, Environmental Noise), Action = Don't worry
2. State = (Sad, Ground, People Talking), Action = Don't worry
3. State = (Neutral, Far, TV/Radio ON), Action = Compliment
4. State = (Happy, Close, Sound Object), Action = Compliment
5. State = (Happy, Close, People Talking), Action = Compliment

In a second moment we ran the model with all 4 features through the same states of the three feature model, of course with the explicit Speech Emotion Recognition value getting:

1. State = (Sad, Sad, Close, Environmental Noise), Action = Don't worry
2. State = (Sad, Neutral, Ground, People Talking), Action = Don't be silly
3. State = (Neutral, Neutral, Far, TV/Radio ON), Action = Compliment
4. State = (Happy, Neutral, Close, Sound Object), Action = Compliment
5. State = (Happy, Happy, Close, People Talking), Action = Compliment

Comparing the result of three state variables with our simulator on four variables we have that in the first model the first state can be interpreted as (Sad, Sad, Close, Environmental Noise) or (Sad, Neutral, Close, Environmental Noise). In order to maximize the reward, the agent will have to carry out actions that bring it respectively to the Neutral-Sad/Sad-Neutral region as regards the first state and to Neutral-Neutral as regards the second. In both cases, the most appropriate action turns out to be Don't worry. The agent therefore did not do wrong action in the three and four feature model.

The second state, on the other hand, can be interpreted in four features such as (Sad, Sad, Ground, People talking) or (Sad, Neutral, Ground, People talking). In this case, the Don't be silly choice is more appropriate, but the agent has decided to perform the Don't worry action in the three feature model, potentially remaining in the Neutral-Sad/Sad-Neutral region. In this case the agent does not take the optimal choice, but still performs an action that does not worsen its situation.
Finally, in the last three states, the agent in both the three and four features model always chooses the Compliment action, an excellent choice to maximize the reward and achieve the goal. Therefore, we can conclude with this experiment that the tree with three features allows us to reduce the complexity of the input by analyzing a lower number of state variables reaching our target in an almost optimal way.

5.5 Results

The three experiments carried out and explained in this chapter end the thesis work developed at the two universities involved in this project. Based on how we designed the environment simulator, the feature importance computed in the first experiment resulted in the two least important state variables being Face Expression Recognition and Speech Emotion Recognition, hypothesis confirmed by calculating the average Monte Carlo error and congruent with the value of the returned policy.
In this context, it was not possible to remove both the two features as the performances obtained in the second experiment were significantly lower than those considering three or four features. For this reason we only removed Speech Emotion Recognition and applied that model on a final demo obtaining results almost equal to those considering all four features.

Chapter 6

Conclusions

6.1 Future Work

As we have already mentioned earlier, one of the most interesting future developments concerning this project is the application of our algorithm on a real robot in order to actually create a social robot that can assist physiotherapists during therapies with children.
At the same time, another aspect of particular interest is to allow the robot to learn online, that is, to generate samples no longer using a simulator, but interacting with the real environment. With this type of learning it would be possible to achieve a Personalized Behavior for each child with whom the robot interfaces making its use more effective. In fact, as we well know, children can have different ages and consequently different behaviors and reactions. For example, a child between 0-5 years old could be calmed by playing a lullaby audio while a more adult child could calm down by listening to a joke or watching a video of a cartoon. With this approach, however, it is necessary to take into account some aspects related to ethics.

In fact, a set of actions that the robot can and cannot perform in particular situations in accordance with the rules of good conduct should be defined, for example the robot cannot hit/slap the child to calm him. In addition, another important point to consider concerns the way in which feedback is given to the robot during both learning and the actual use of the robot itself, a possible solution could be to use a physiotherapist who can communicate the reward to the robot it deserves after having a certain behavior. Of course, the entire development process is complex, therefore further aspects must be taken into consideration. Not forgetting that the results obtained and new future investigations could be used not only in the field of rehabilitation, but in all those social contexts in which the robot could integrate and collaborate with a specialist in order to simplify the latter's work.

Bibliography

[1] Ahmed Bahgat El Seddawy, D. A. K., Prof. Dr Turkey Sultan. Applying classification technique using did3 algorithm to improve decision support under uncertain situations. *International Journal of Modern Engineering Research* , (2013), 2139.

[2] Bellemare, M. G., Dabney, W., Dadashi, R., Taïga, A. A., Castro, P. S., Roux, N. L., Schuurmans, D., Lattimore, T., and Lyle, C. A geometric perspective on optimal representations for reinforcement learning. *CoRR*, **abs/1901.11530** (2019). Available from: `http://arxiv.org/abs/1901.11530`, `arXiv:1901.11530`.

[3] Bernstein, D., Crowley, K., and Nourbakhsh, I. Working with a robot: Exploring relationship potential in human–robot systems. *Interaction Studies*, **8** (2007), 465.

[4] Bradski, G. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, (2000).

[5] Breiman, L. Classification and regression trees. (1984).

[6] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym (2016). Cite arxiv:1606.01540. Available from: `http://arxiv.org/abs/1606.01540`.

[7] Chollet, F. keras. `https://github.com/fchollet/keras` (2015).

[8] Dadashi, R., Taïga, A. A., Roux, N. L., Schuurmans, D., and Bellemare, M. G. The value function polytope in reinforcement learning. *CoRR*, **abs/1901.11524** (2019). Available from: `http://arxiv.org/abs/1901.11524`, `arXiv:1901.11524`.

[9] de Pinto, M. G. Emotion-classification-ravdess. Available from: `https://github.com/marcogdepinto/Emotion-Classification-Ravdess`.

[10] Faggella, D. What is machine learning? Available from: `https://emerj.com/ai-glossary-terms/what-is-machine-learning/`.

[11] for Visual Recognition, C. C. N. N. Convolutional neural networks (cnns / convnets). Available from: `http://cs231n.github.io/convolutional-networks/`.

[12] Iocchi, L. Machine learning course. (2018/2019).

[13] Kaggle. Challenges in representation learning: Facial expression recognition challenge. Available from: `http://tiny.cc/96kciz`.

[14] Kahn, P. H., Ishiguro, H., Friedman, B., and Kanda, T. What is a human? - toward psychological benchmarks in the field of human-robot interaction. In *ROMAN 2006 - The 15th IEEE International Symposium on Robot and Human Interactive Communication*, pp. 364–371 (2006). `doi:10.1109/ROMAN.2006.314461`.

[15] Kuo, C. J. Understanding convolutional neural networks with A mathematical model. *CoRR*, **abs/1609.04112** (2016). Available from: `http://arxiv.org/abs/1609.04112`, `arXiv:1609.04112`.

[16] Livingstone, S. and Russo, F. The ryerson audio-visual database of emotional speech and song (ravdess): A dynamic, multimodal set of facial and vocal expressions in north american english. *PLOS ONE*, **13** (2018), e0196391. `doi:10.1371/journal.pone.0196391`.

[17] Lo Presti, A. Master's thesis github repository: Machine learning for social assistive robots. Available from: `https://github.com/alessandrolopresti/master-thesis/tree/master/thesis`.

[18] M Mazid, M., M Shawkat Ali, A. B., and Tickle, K. Improved c4.5 algorithm for rule based classification.

[19] Madnani, M. Fer - facial expression recognition. Available from: `https://github.com/mayurmadnani/fer`.

[20] Medium. Available from: `https://medium.com/`.

[21] Pedregosa, F., et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, **12** (2011), 2825.

[22] Rajalingam, A., Matharu, D., Vinayagamoorthy, K., and Ghoman, N. Data mining course project: Income analysis. *SFWR ENG/COM SCI 4TF3*, (December 10, 2002).

[23] Redmon, J., Divvala, S. K., Girshick, R. B., and Farhadi, A. You only look once: Unified, real-time object detection. *CoRR*, **abs/1506.02640** (2015). Available from: `http://arxiv.org/abs/1506.02640`, `arXiv:1506.02640`.

[24] Ronaghan, S. The mathematics of decision trees, random forest and feature importance in scikit-learn and spark. Available from: `https://bit.ly/37LZME0`.

[25] Ross Quinlan, J. Induction of decision trees. *Machine learning*, **1** (1986). Available from: `https://link.springer.com/article/10.1007/BF00116251`.

[26] Ross Quinlan, J. C4.5: Program for machine learning. (1993). Available from: `https://ci.nii.ac.jp/naid/10015645285/`.

[27] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *CoRR*, **abs/1707.06347** (2017). Available from: `http://arxiv.org/abs/1707.06347`, `arXiv:1707.06347`.

[28] Silver, D. UCL course on reinforcement learning. (2015).

[29] Simonini, T. An intro to advantage actor critic methods: let's play sonic the hedgehog! Available from: `https://bit.ly/2FxIxdw`.

[30] Simonini, T. An introduction to policy gradients with cartpole and doom. Available from: `https://bit.ly/35ErEs1`.

[31] Singh, S. and Gupta, P. Comparative study id3, cart and c4.5 decision tree algorithm: A survey. *IJAIST*, **27** (2014). Available from: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.685.4929&rep=rep1&type=pdf`.

[32] Singh, S. P. and Sutton, R. S. Reinforcement learning with replacing eligibility traces. *Machine Learning*, **22** (1996), 123. Available from: `https://doi.org/10.1007/BF00114726`, `doi:10.1007/BF00114726`.

[33] Sphere Software. Machine learning made more effective through python. Available from: `https://www.sphereinc.com/machine-learning-made-more-effective-through-python/`.

[34] stackoverflow. What is the number of filter in cnn? Available from: `https://stackoverflow.com/questions/36243536/what-is-the-number-of-filter-in-cnn`.

[35] Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. The MIT Press, second edn. (2018). Available from: `http://incompleteideas.net/book/the-book-2nd.html`.

[36] van Rossum, G. Python tutorial. Tech. Rep. CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1995).

[37] Watkins, C. J. C. H. and Dayan, P. Q-learning. *Machine Learning*, **8** (1992), 279. Available from: `https://doi.org/10.1007/BF00992698`, `doi:10.1007/BF00992698`.

[38] Wikipedia the free encyclopedia. Available from: `https://www.wikipedia.org/`.

[39] Yale Universiy. Socially assistive robotics: An nsf expedition in computing. Available from: `https://robotshelpingkids.yale.edu/overview`.

[40] Zhou, Z.-H. and Feng, J. Deep forest: Towards an alternative to deep neural networks. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, pp. 3553–3559. AAAI Press (2017). ISBN 978-0-9992411-0-3. Available from: `http://dl.acm.org/citation.cfm?id=3172077.3172386`.