Alma Mater Studiorum · University of Bologna

School of Science
Department of Physics and Astronomy
Master Degree in Physics

Deep Reinforcement Learning for Industrial Applications

Supervisor:
Prof. Nico Lanconelli

Co-supervisors:
Prof. Renato Campanini
Dr. Matteo Roffilli

Submitted by:
Tommaso Mariani

Academic Year 2018/2019

Abstract

In recent years there has been a growing attention from the world of research and companies in the field of Machine Learning. This interest, thanks mainly to the increasing availability of large amounts of data, and the respective strengthening of the hardware sector useful for their analysis, has led to the birth of Deep Learning. The growing computing capacity and the use of mathematical optimization techniques, already studied in depth but with few applications due to a low computational power, have then allowed the development of a new approach called Reinforcement Learning.

This thesis work is part of an industrial process of selection of fruit for sale, thanks to the identification and classification of any defects present on it. The final objective is to measure the similarity between them, being able to identify and link them together, even if coming from optical acquisitions obtained at different time steps.

We therefore studied a class of algorithms characteristic of Reinforcement Learning, the policy gradient methods, in order to train a feedforward neural network to compare possible malformations of the same fruit. Finally, an applicability study was made, based on real data, in which the model was compared on different fruit rolling dynamics and with different versions of the network.

Sommario

Negli ultimi anni si è assistito ad una crescente attenzione da parte del mondo della ricerca e delle aziende al campo del Machine Learning. Questo interessamento, grazie soprattutto alla crescente disponibilità di grandi quantità di dati, e dal rispettivo potenziamento del comparto hardware utile alla loro analisi, è sfociato nella nascita del Deep Learning. La crescente capacità di calcolo e l'utilizzo di tecniche di ottimizzazione matematica, già in precedenza studiate approfonditamente ma con poche applicazioni a causa della scarsa potenza computazionale, hanno poi permesso lo sviluppo di un nuovo approccio che prende il nome di Reinforcement Learning.

Questo lavoro di tesi si inserisce all'interno di un processo industriale di selezione della frutta destinata alla vendita, grazie all'individuazione ed alla classificazione di eventuali difetti presenti su di essa. L'obiettivo finale è quello di essere in grado di misurare la similarità tra questi, riuscendo ad identificarli ed a collegarli tra loro, anche se provenienti da acquisizioni ottiche ottenute ad istanti temporali differenti.

Si è quindi studiato una classe di algoritmi caratteristici del Reinforcement Learning, i policy gradient methods, al fine di addestrare una rete neurale feedforward a confrontare tra loro le eventuali malformazioni di uno stesso frutto. Infine si è fatto uno studio di applicabilità, basato su dati reali, in cui si è confrontato il modello su diverse dinamiche di rotolamento dei frutti e con differenti versioni della rete.

Ai miei genitori

Acknowledgements

I would first like to thank my thesis supervisor Prof. Nico Lanconelli of the Dept. of Physics and Astronomy at University of Bologna, for his assistance, his time, and his valuable advice on the development of this thesis work.

I would also like to express my gratitude to Prof. Renato Campanini of the Dept. of Physics and Astronomy at University of Bologna, for his lessons, which allowed me to discover and love the field of machine learning, and for his willingness to listen, which helped me to solve many doubts and questions.

Special thanks undoubtedly go to Dr. Matteo Roffilli, CEO of Bioretics s.r.l., for his constant willingness to share his knowledge with a constructive dialogue, which made this thesis work possible. An acknowledgement also goes to Ser.mac s.r.l., which made available the fruits dataset needed for the final part of this project.

Finally, I would like to recognize the invaluable moral support and continued encouragement received from my family, friends, and colleagues here at the University of Bologna. Each of you, in your own unique way, has always pushed me to do my best, and I hope I have made you proud of me.

Contents

List of Algorithms

List of Figures

List of Tables

General Introduction

In Chapter 1, basic machine learning concepts are introduced, the different learning approaches, what does it mean to overfit a model, the importance of hyperparameters and their role, and the gradient-based optimization and how it is implemented.

In Chapter 2, a very common model for deep learning, feedforward neural networks, is presented and the different parts from which it is composed of, such as perceptrons, layers, activation functions, and the backpropagation algorithm, are described in details.

In Chapter 3, the reinforcement learning framework is introduced, with particular attention to a formal definition of finite Markov decision processes, and the fundamental elements of reinforcement learning, such as states, actions, rewards, and policies.

In Chapter 4, some basic reinforcement learning algorithms are presented, useful for understanding general ideas such as generalized policy iteration, which describes the relationship between the value function and the policy.

In Chapter 5, the concepts presented before are extended thanks to the use of functions approximation, which allows solutions to be found even in a more general context, allowing the treatment of more complex problems.

In Chapter 6, the problem faced is described and an alternative approach is proposed; the methods and models actually used to solve this are then presented, with particular regards to the fruits modelling process and data generation.

In Chapter 7, several tests carried out on the dataset are presented, both as regards the study of different dynamics and the comparison between different models; the numerical results obtained are then reported, from which some conclusions can be drawn, also with a view to future developments.

Part I

Deep Learning

Chapter 1

Introduction to Machine Learning

1.1 Basic Concepts

Today the word *artificial intelligence* is used, and often abused, in a large number of areas, both scientific or not, and it is good to clarify what we are actually talking about. Practically speaking, AI is a thriving field with many practical applications and active research topics. Researchers look for intelligent software to automate routine labor, understand speech or images, make diagnoses in medicine and support basic scientific research. The true challenge to artificial intelligence is in fact in solving tasks that are easy for people to perform, but hard to describe formally [Goodfellow et al., 2017].

The solution might be to allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined in terms of its relation to simpler concepts. This hierarchy allows the computer to learn complicated concepts by building them out of simpler ones. If this process were represented graphically it would probably be similar to a graph showing how all these ideas are built on top of each other, with the graph being deep, with many layers. For this reason, this approach to AI is usually called *deep learning*.

The difficulties faced by systems relying on coded knowledge suggest that deep learning systems need the ability to acquire their own knowledge, by extracting patterns from raw data. This capability is known as *machine learning*. The performance of these machine learning algorithms depends heavily on the *representation* of the data they are given. Each piece of information included in this representation is known as a *feature*.

1.1.1 Task, Performance and Experience

In [Mitchell, 1997] it is very clearly defined what is meant by *learning* for a computer:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

In this relatively formal definition of the word "task", it is important to notice that the process of learning itself is not the task. Learning is the means of attaining the ability to perform the task T. Machine learning tasks are usually described in terms of how the machine learning system should process an *example*. An example is a collection of features that have been quantitatively measured from some object or event. An example is generally represented as a feature vector $\boldsymbol{x} \in \mathbb{R}^d$, where each entry x_i is a different feature. Several are the tasks that a machine learning algorithm can perform, such as regression, classification, probability density estimation and many others.

In order to evaluate the ability of a machine learning algorithm to perform such a task, there is need to define a quantitative way of defining a measure of its performance P, usually specific to the task T being carried on. For tasks such as classification and so on, often the *accuracy* of the model is measured, or the proportion of examples for which the model produces the correct output. Equivalent information can be obtained by measuring the *error rate*, the proportion of examples for which the model produces and incorrect output. It is important to say that only the performance of data that the model has not seen before is interesting, since this can an esteem of how well it would work when deployed in the real world.

Supervised vs. Unsupervised Learning

Machine learning algorithms can also be broadly categorized as *unsupervised* or *supervised* based upon what kind of experience they are allowed to have during the learning process and how the training dataset is structured. Unsupervised learning algorithms usually have a dataset containing many features and then learn the useful properties of the structure of this dataset. Supervised learning algorithms, on the other hand, have a dataset containing features too, but each of these is also associated with a *label* (or target). Roughly speaking, unsupervised learning involves observing several examples of an input vector $\boldsymbol{x}$ and trying to learn some interesting properties of the underlying

distribution $p(\boldsymbol{x})$, while supervised learning involves observing several examples of the same vector and of the associated value vector $\boldsymbol{y}$, learning to predict $p(\boldsymbol{y}|\boldsymbol{x})$.

1.1.2 Flexibility, Underfitting and Overfitting

As said before, the main challenge of deep learning is to perform well on new, unseen data. This ability is called *generalization*. When a machine learning model is in training, it is possible to compute some error measure of the training set called *training error*. But actually we want to minimize the so called *generalization error* (often also called *test error*), which is defined as the expected value of the error on a new input. This is typically estimated by measuring the performance on a test set of examples, which are kept separated from the training set. The machine learning process can then be summarized in two steps: the training set is sampled and the model is trained to minimize training error; and then it is tested on a test set to determine the generalization error.

As a consequence of the entire process, the expected test error is ofter greater or equal than the training error. The factors determining how well a machine learning algorithm will perform are its ability to make the training error the smallest as possible and shrinking the gap between the test error. These two factors corresponds to the two central challenges in machine learning: *underfitting* and *overfitting*. The first occurs when the model is not able to obtain a sufficiently low error value on the training set, whilst the second occurs when the gap between the two errors is too large. A graphical representation of the two is reported in Figure 1.1.

Is it possible to control how much likely a model will overfit or underfit by altering its property called *flexibility*. Practically speaking, a model's flexibility is its ability to fit a wide variety of functions. Flexible models can overfit by simply memorizing the correct answers to the training set, while models with low flexibility may have some difficulties to fit it. One way to control the flexibility of the model is to choose the dimension of its *hypotesis space*, the set of functions that the learning algorithm is allowed to select as the general solution of the problem.

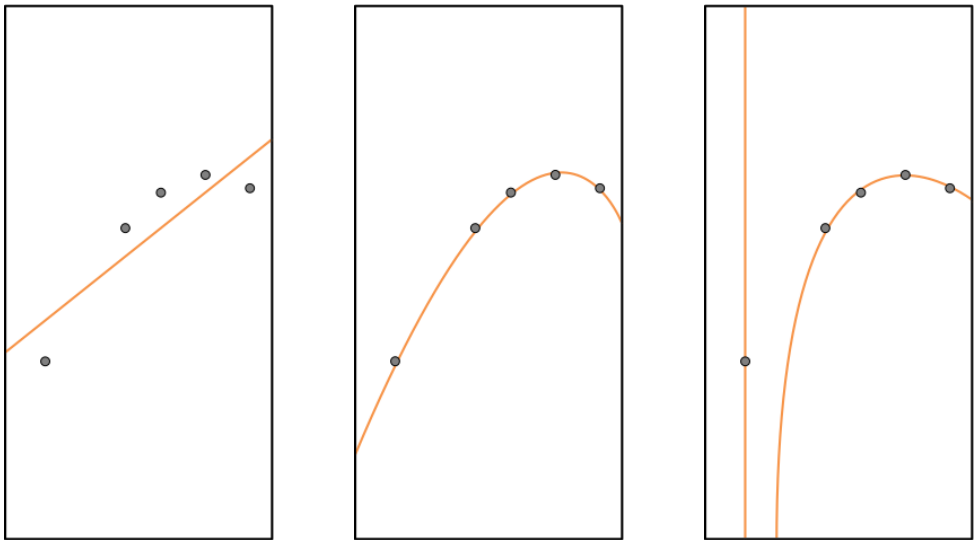
For example, the flexibility of a linear regression spline, which is represented by:

$$\hat{y} = b + \sum_{i=1}^{n} w_i x^i \tag{1.1}$$

it's in the number of their parameters w_i (and, as a consequence, in the maximum grade of the polynomial). However, it is important to note that, even if there are polynomials

of high grade in the function, this is still linear since the equation is a linear combination of the parameters w_i.

Machine learning algorithms generally perform best when their flexibility is appropriate for the true complexity of the task they need to perform. Models with insufficient flexibility are obviously unable to solve complex tasks. On the other hand, models with high flexibility can solve complex tasks, but when their flexibility is too high for the currentt task they may overfit. These two situations are graphically represented as an example in Figure 1.1, where the simple linear regression on the left side does not capture the complexity of the system below, while the higher degree model on the right side is too precise and will probably have an higher value of the test error.



(a) Underfitting (b) Appropriate (c) Overfitting

Figure 1.1: Flexibility of a model.

A graphical representation of the struggle to find the best combination between underfitting and overfitting is also reported in Figure 1.2. In this image are depicted the trends of the training error and the test error with respect to the flexibility of the model and the conncection with Figure 1.1 is immediately clear. A low flexibility in the model would probably lead to a bounded *generalization gap* (difference of the test and training error), but also to a high test error, due to the model's inability to generalize. On the other hand, a high flexibility of the model will bring a lower test error in a first instance but will probably lead to overfitting, causing a diverging generalization gap.

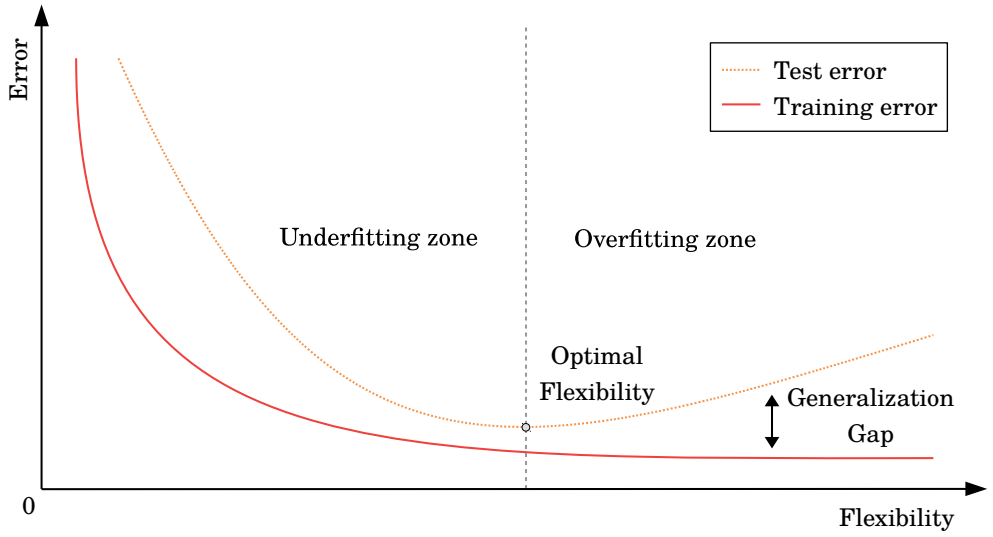


Figure 1.2: Optimal model flexibility.

1.1.3 Validation Set, Hyperparameters, and Cross-Validation

An easy but effective strategy to avoid overfitting is to define another set, called *validation set*, from the training set containing samples which the training set that will not be observed during the training phase (this is made because the test set cannot be used in any other way than to estimate the error rate), against which we are going to evaluate model's performance. Assuming that these two datasets are indipendently extracted and identically distributed, the optimization goal becomes lowering the train and validation errors while ensuring that the difference between the two stays low.

Measuring the performance on both the datasets, during the training phase, will show that performance improves for both the errors until the model starts to reproduce the features of the training dataset. In this case we observe that the train performance continues to slowly decrease as expected, while the validation error increases as the model starts to overfit. Interrupting the training process before the validation error starts to increase should ensure that the model optimizes while retaining enough generalization capabilities, this trick is called *early stopping*. Since we're not evaluating the actual performance on the test set, there is no precise point where we need to stop training but rather an interval, as depicted in Figure 1.3.

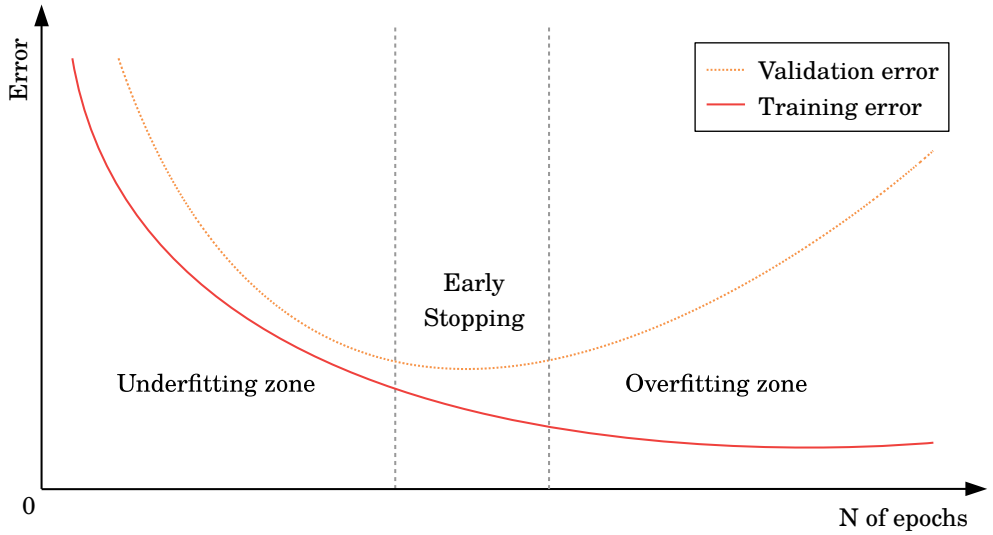


Figure 1.3: Early stopping.

Most machine learning algorithms also have the possibility to control some parameters, which are called *hyperparameters*, in order to modify the general behaviour of the model. In the linear regression example from Equation 1.1, the only hyperparameter is the degree of the polynomial, but often there are more of them. The values of hyperparameters are usually not learned through the learning algorithm and, since this can be viewed as an underfitting vs. overfitting problem, it is usally used the validation set for this purpose too. It is important to notice that, as explained before, the validation error will of course underestimate the generalization error, though typically by a smaller amount than the training error.

The main problem now, is how to divide the dataset into the training plus validation part and the test part. This is crucial especially when the dataset is small, since this implies a big uncertainty around the estimated test error. The most common technique for solving the problem is called *k-fold Cross-Validation*, which allows to use all the examples for the estimation of the generalizaion error at the cost of increased computation. This procedure is based upon the idea of repeating the training and testing on differently random chosen splits of the original dataset. The dataset is divided into k non-overlapping subsets and, on trial i, the i-th subset of the data is used to estimate the test set, while the rest of the data is used as the training and validation sets.

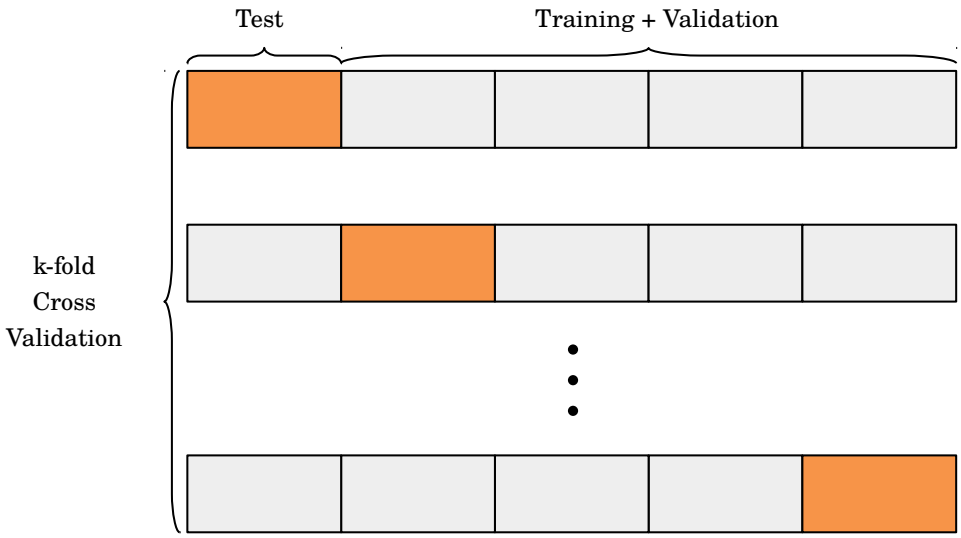


Figure 1.4: K-fold Cross Validation.

1.2 Gradient-Based Optimization

As a consequence of what has being said about minimizing the error functions, it is possible to think deep learning algorithms as *optimization problems*. In optimiziation problems, one tries to minimize or maximize some function $f(x)$, the so called *objective function*, or criterion, by modifying the input x. In the deep learning framework, this function is obviously the training error, which is often also called *cost function*, *error function*, or simply the *loss* of the model.

The most common technique in optimization problems is the *gradient descent*, which exploits the properties of the first derivative $f'(x)$ of the loss function to find its minimum. The derivation operation is very useful in this task, since it tells us how to change the input x in order to slightly modify $f(x)$ in the direction we want. In fact, it is immediate to demonstrate that $f(x - \epsilon \operatorname{sign}(f'(x)))$ is less than $f(x)$ for a small enough ϵ.

For functions with several variables it is necessary to introduce *partial derivatives*. The partial derivative $\frac{\partial}{\partial x_i} f(\boldsymbol{x})$ measures how f changes as only the variable x_i increase at point $\boldsymbol{x}$. The vector containing all the partial derivatives, with respect to different directions, is called *gradient* and denoted by $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$. This generalizes the concept of derivative in multiple dimensions and therefore a critical point has a zero gradient.

The points in which $f'(x) = 0$, the derivative provides no information about which direction leads to an increase, or a decrease, of the function. Such points are called *critical*, or stationary, points. Each of these is also called a *local minimum* if $f(x)$ is lower than all neighboring points, or a local maximum if higher. If it is neither a minimum or a maximum, it is called a *saddle point*. If a point has also the absolute lowest value of $f(x)$, it is called the *global minimum*, or global maximum if it is the highest.

In the context of deep learning, is happens very often to have functions with many local minima wich are not optimal, and many saddle points surrounded by flat regions. Since this makes the learning process very difficult and slow, it's usually fine to settle for finding a value of $f(x)$ that is very low, but not necessarily minimal. An example of a objective function trend is reported in Figure 1.5.

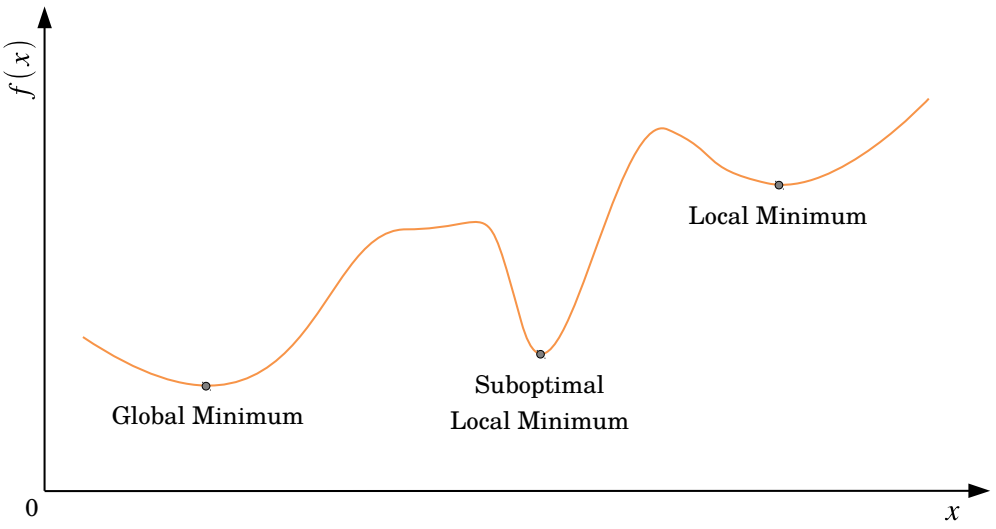


Figure 1.5: Minima optimization landscape.

1.2.1 Minibatch Stochastic Gradient Descent

The gradient descent technique is based on the property of the gradient to always point in the direction of maximum increase of the objective function $f(\boldsymbol{x})$. From this, we deduce that it is possible to decrease the function by simply moving in the direction of the negative gradient $-\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$.

This method, known as method of steepest descent, or simply *gradient descent*, proposes a new point $\boldsymbol{x}'$, for which $f(\boldsymbol{x}') < f(\boldsymbol{x})$:

$$\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \tag{1.2}$$

where ϵ is a positive valued parameter, called *learning rate*.

In terms of deep learning, the objective function to minimize is usually the loss function $L(\boldsymbol{x}, \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is the vector that describes the model's paramters. The gradient is then taken with respect to $\boldsymbol{\theta}$, in order to reduce the loss over the input values $\boldsymbol{x}^{(i)}$ from the dataset. If a gradient step is evaluated over all the training set, the algorithm is said to be *deterministic* but, if the dataset dimension n is very large, it can be too computationally expensive, and the time to take a gradient step becomes prohibitively long. On the other hand, if a gradient step is evaluated for each of the dataset samples the method is called *stochastic*, but the result is probably very noisy.

Most algorithms used for deep learning fall somewhere in between, using more than one but less than all of the training samples, and are then called *minibatch* methods. The novelty introduced by these is that the gradient descent is now based upon an expectation, which is approximately estimated using a subsample of the dataset. On each step of the algorithm in fact, a batch $\mathcal{B} = \{\boldsymbol{x}^{(1)} \dots \boldsymbol{x}^{(m)}\}$ of examples is fed from the dataset into the algorithm. Here the batch size m is typically chosen to be relatively small with compared to the dataset size n. An implementation of the resulting algorithm is reported in Algorithm 1.

Algorithm 1: Minibatch Stochastic Gradient Descent

Inputs: a training set $\boldsymbol{X}$, a loss function $L(\boldsymbol{\theta}, \boldsymbol{x})$
Parameters: model parameters $\boldsymbol{\theta}$, learning rate ϵ, a small threshold η used to determine the accuracy of the parameters $\boldsymbol{\theta}$
Initialization: θ_i randomly

while $L(\boldsymbol{\theta}, \boldsymbol{x}) < \eta$ **do**
  sample $\{\boldsymbol{x}^{(1)} \dots \boldsymbol{x}^{(m)}\}$ from $\boldsymbol{X}$
  $\boldsymbol{g} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} L(\boldsymbol{x}^{(i)}, \boldsymbol{\theta})$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g}$
return $\boldsymbol{\theta}$

1.2.2 Adaptive Learning Rates

A crucial parameter of the gradient descent algorithm is the learning rate ϵ, and it is usually very difficult to set because it a has a significant effect of model performance. In fact, it is important to decrease ϵ over time because, by random sampling the m training samples, a lot of noise is introduced in the process, which does not vanish when arriving at a minimum. Initially it was common to decay the learning rate linearly until a certain iteration τ, after which it will simply remains constant:

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau \tag{1.3}$$

with $\alpha = \frac{k}{\tau}$. This method is certainly simple and effective in making the algorithm converge but it can be improved quite easily.

The easisest way to do so is by introducing *momentums*, which are designed to accelerate learning, especially when facing small but consistent gradients. Formally, this algorithm introduces a variable $\boldsymbol{v}$, that plays the role of a sort of velocity of the parameters. This velocity is set to an exponentially decaying average of the negative gradients. An hyperparameter $\alpha \in [0, 1]$ determines how quickly the contributions of previous gradients exponentially decay, and the new update rule is then given by:

$$\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \epsilon\nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}, \boldsymbol{\theta})) \right) \tag{1.4}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v} \tag{1.5}$$

Here, the velocity $\boldsymbol{v}$ accumulates the gradients elements $\nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}, \boldsymbol{\theta})) \right)$ and, the larger α is relative to ϵ, the more previous gradients affect the current direction.

Actually it is possible to think the problems of finding the best learning parameters and the best learning rate as separated, and to adapt a different learning rate for each parameter. The *AdaGrad* algorithm from [Duchi et al., 2011], for example, individually adapts the learning rates of all parameters by scaling them inversely proportional to the square root of the sum of all of their historical squared values. The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate. The final effect is greater progress in the lesser sloped directions of parameter space.

A slightly modification of this is the *RMSProp* algorithm, from [Hinton, 2012], which changes the gradient accumulation into an exponential weighted moving average. This

is done to discard history form the extreme past so the model can converge rapidly after finding a convex bowl. This algorithm has been shown to be an effective and practical optimization algorithm for neural networks, and its one of the best optimization algorithms available.

The last learning rate optimization algorithm presented is *Adam* (which stands for "adaptive moments"), from [Kingma and Ba, 2014], which is a variant of the combination of the RMSProp with momentum. First, the momentum is incorporated directly as an estimate of the first order moment (with exponential weighting) of the rescaled gradients. Second, first order and second order moments estimates are bias corrected. The implementation of the stochastic gradient descent algorithm with the Adam optimizer is reported in Algorithm 2, where the $\odot$ operation is intended as an element-wise product.

Algorithm 2: Minibatch Stochastic Gradient Descent (Adam Optimizer)

Inputs: a training set $\boldsymbol{X}$, a loss function $L(\boldsymbol{\theta}, \boldsymbol{x})$

Parameters: function parameters $\boldsymbol{\theta}$, learning rate ϵ (usually 10^{-4}), exponential decay rates ρ_1 and ρ_2 (usually 0.9 and 0.999 respectively), a small constant δ for numerical stabilization (usually 10^{-8}), a small threshold η used to determine the accuracy of the paramters $\boldsymbol{\theta}$

Initialization: θ_i randomly, first and second moment variables $\boldsymbol{s} = 0$ and $\boldsymbol{r} = 0$, timestep $t = 0$

while $L(\boldsymbol{\theta}, \boldsymbol{x}) < \eta$ **do**

- sample $\{\boldsymbol{x}^{(1)} \dots \boldsymbol{x}^{(m)}\}$ from $\boldsymbol{X}$
- $\boldsymbol{g} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} L(\boldsymbol{x}^{(i)}, \boldsymbol{\theta})$
- $t \leftarrow t + 1$
- biased first moment estsimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
- biased second moment estsimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
- correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
- correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
- $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$

return $\boldsymbol{\theta}$

Chapter 2

Artificial Neural Networks and Deep Learning

2.1 Introduction

Deep feedforward neural networks, often simply called deep neural networks, are the representative model for deep learning. The main goal of a deep neural network is to approximate some function $f^*(\boldsymbol{x})$, for example a classifier which maps an input vector $\boldsymbol{x}$ to a certain class y_i, coded inside the class vector $\boldsymbol{y}$. From a mathematical point of view, a deep neural network defines a mapping:

$$\boldsymbol{y} = f(\boldsymbol{x}, \boldsymbol{\theta}) \tag{2.1}$$

where the parameters $\boldsymbol{\theta}$ are the trainable parameters, which are learned during the training phase, and which lead to the best function approximation.

These models are called *feedforward* because the information travels only in a certain direction, from the input $\boldsymbol{x}$, directly to the output $\boldsymbol{y}$. Unlike in *recurrent* neural networks (RNN), not covered here, there are no feedback connections, in which outputs of the model are taken back as input into themselves.

Feedforward neural networks are also called *networks*, because they are typically associated with a directed acyclic graph, describing how the different parts are composed together. In Figure 2.1 a simple network is represented, in which three function $f^{(1)}, f^{(2)}, f^{(3)}$ are linked together to form the composed function:

$$f = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x}))) \tag{2.2}$$

In this case, each function is called a *layer* of the network, i.e. $f^{(1)}$ if the first layer, $f^{(2)}$ if the second layer, and so on. The final layer of the network is called the *output layer*, while the overall length of the chain is named the *depth* of the model. Since only the output is compared to the expected result $y = f^*(\boldsymbol{x})$, the layers in between are called *hidden layers*. Each of these is typically vector-valued, and the dimensionality of the largest one determines the *width* of the model. During the training phase of the neural network, the learning algorithm decides how to tune these hidden layers, in order to implement the best approximation of f^*.

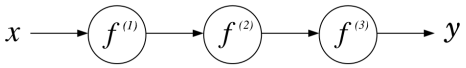


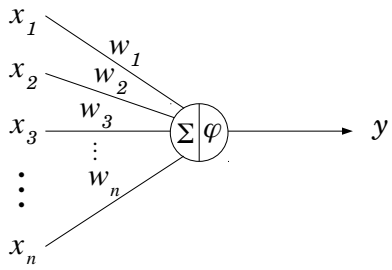
Figure 2.1: Basic feedforward network.

Finally, these networks are said to be *neural* because the loosely resembles a network of neurons from neuroscience. Each element of the layer is called a *perceptron*, hence the alternative name for neural networks: multilayer perceptrons (MLP), which are thought to play a role similar to a neuron. Thanks to this similarity, it is possible to think a single layer as consisting of many units that act in parallel, each representing a vector-to-scalar function rather than a vector-to-vector function, which, as we shall see, leads to simplifications in the application of gradient descent algorithms.

2.2 Single Layer Perceptron

As a consequence of what has been said, a single perceptron takes a vector-valued input and outputs a scalar. In order to do this, each unit has an unique *weights vector* $\boldsymbol{w} = \{w_1, w_2, \ldots, w_p\}$, each of which will be updated and learned during the training phase of the model, and an *activation function* $\varphi(x)$, usually non-linear. The perceptron behaviour can then be divided into two steps: firstly, the perceptron computes the weighted sum of the input vector $\sum_j w_j x_j$ and, secondly, in applies the activation function so the obtained result. The perceptron is also said to be linear or non-linear, depending on the nature of its activation function.

Thanks to its neural nature, its quite common to represent a generic perceptron as depicted in Figure 2.2, alongside with its mathematical equation.

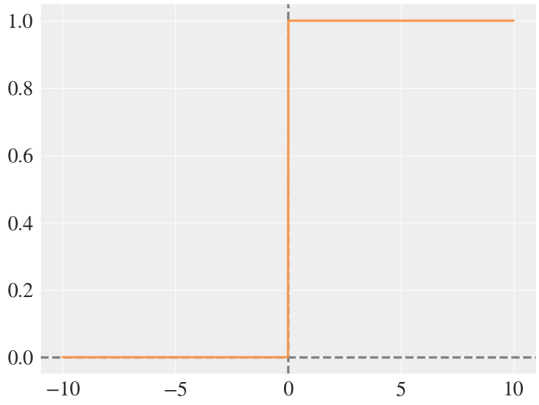


$$y = \varphi\left(\sum_j x_j w_j\right) \tag{2.3}$$

Figure 2.2: Single layer perceptron.

2.2.1 Linear Case

Of all the possible cases, the simplest one is undoubtedly the one in which the activation function is a linear one. One of the simplest activation functions available in this case is the *binary* activation function (also called Heaviside step function):



$$\varphi(x) = H(x) \doteq \begin{cases} 1, \text{ if } x \geq \theta \\ 0, \text{ if } x < \theta \end{cases} \tag{2.4}$$

Figure 2.3: Binary activation function.

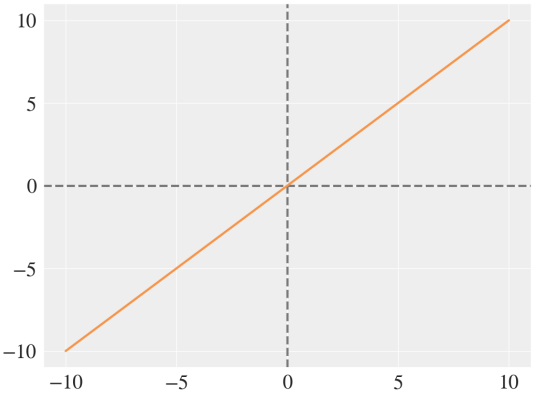
In Figure 2.3 the parameter was fixed $\theta = 0$ for the simplicity of representation, but in general it can be fixed to any value. The main problem with a step function is that it doesn't allow vector-value outputs (e.g. it doesn't support the classification of the inputs into one of several categories), or at least it would need one additional parameter θ_i for each additional class, rising up the number of the parameters to choose (and therefore the flexibility of the model).

Despite being so simple, it is possible to describe some boolean functions like AND or OR using a perceptron without weights and a binary step function. In order to do so threshold should be $\theta = 2$ and $\theta = 1$, for the AND and for the OR function respectively, with the inputs that are only between $x_i \in \{0, 1\}$.

(a) AND (b) OR

Figure 2.4: Neural implementation of logic gates.

Another functions often used is the simple *linear* activation function:



$$\varphi(x) = \alpha \cdot x \tag{2.5}$$

Figure 2.5: Linear activation function.

which takes the weighted sum of the inputs and creates an output signal directly proportional to it. In a certain sense, a linear function is better than a step function because it allows multiple outputs without incrementing the number of parameters. As before, in Figure 2.5 the parameter was fixed $c = 1$ for the simplicity of representation, but in general it can be fixed to any value.

Linear perceptrons, however, are not really useful in more advanced tasks, like image recognition or classification. This is due to the fact that it is not possible to use a technique called *backpropagation* (based on the gradient descent method), which will be explained later, to train the model. The derivative of a linear function is in fact

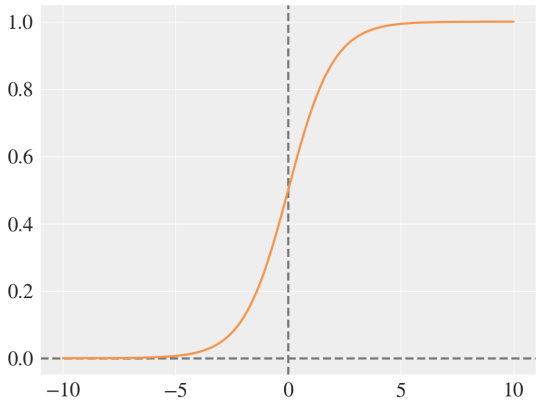
constant, and therefore has no relation to the input vector $\boldsymbol{x}$, so it is not possible to go back and understand which weights or parameters in the input neurons can provide better prediction. Another reason is that, with linear activations, all the hidden layers of the neural network collapse into one. No matter how many there are, the last layer will be a simple linear function of the first (since a combination of linear functions is still a linear functions). This prevents the model from stacking several layers and to achieve a more complex behaviour.

2.2.2 Non-linear Case

Non-linear perceptrons differently, uses non-linear activation functions and, therefore, can be stacked to form a deeper network with more complex behaviour. Actually, there are a lot of different functions to choose from in the non-linear case, each one with some pros and cons. It is also important to distinguish between the ones which can be used in the output layer and the ones used mainly in the hidden ones.

Output Units

Many tasks often requires to predict the probability of an input vector $\boldsymbol{x}$ to belong to class y_i. Since the output must be bounded in the interval $[0, 1]$, it's useful to use non-linear functions with the same span. One of the most common non-linear functions with this property, is the *sigmoid* activation function, defined as:



$$\varphi(x) = \sigma(x) \doteq \frac{1}{1 + \exp(-x)} \qquad (2.6)$$

Figure 2.6: Sigmoid activation function.

which normalizes the output in the interval $[0, 1]$. This function is very convenient because it provides a smooth gradient, without jumps in output values, and leads to

clear predictions, since even a relatively small value brings the output close to 0 or 1. On the other hand, for very high or very low values of x, there is almost no change to the output y, causing the *vanishing gradient problem*. This can result in the network refusing to learn further, or being too slow to reach an accurate prediction.

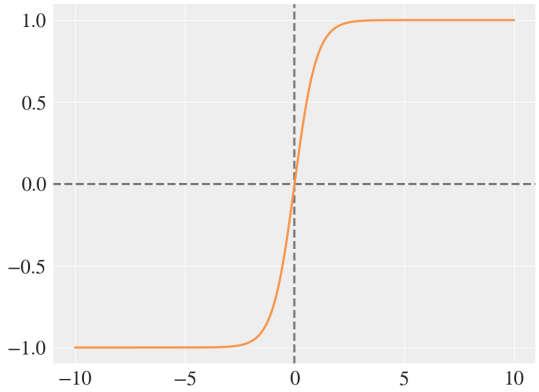
The immediate extension of the sigmoid, which is capable to handle multiple classes, is the *softmax activation function*:

$$\varphi(x) = \text{softmax}(x_i) \doteq \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{2.7}$$

which normalizes the outputs for each class between 0 and 1, and divides by their sum, giving the probability of the input value being in a specific class. This function is very useful in multi-class problems, rather than using several sigmoids, because it also enhances the differences between probabilities, but ensuring that the sum is always 1.

Hidden Units

Prior to the introduction of rectified linear units (which are described in a while), most of the neural network models used the sigmoid $\sigma(x)$ also for the hidden layers or, alternatively the *hyperbolic tangent* activation function:

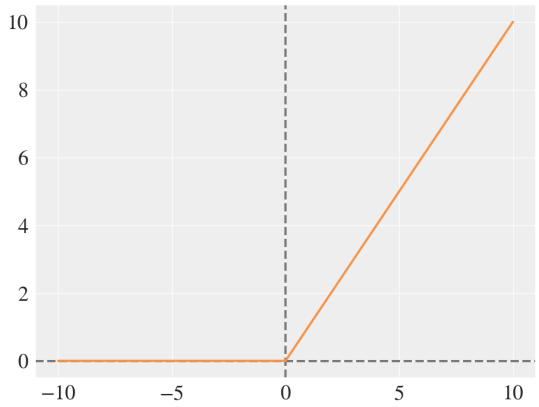


$$\begin{aligned} \varphi(x) &= \tanh(x) \\ \tanh(x) &\doteq \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \end{aligned} \tag{2.8}$$

Figure 2.7: Hyperbolic tangent activation function.

which is closely related to the sigmoid from the relation $\tanh(x) = 2\sigma(2x) - 1$. But, apart from this, the hyperbolic tangent is quite different from the sigmoid, since now the output is zero centered and bounded to be in $[-1, 1]$, which is important because it allows the activation to also be negative.

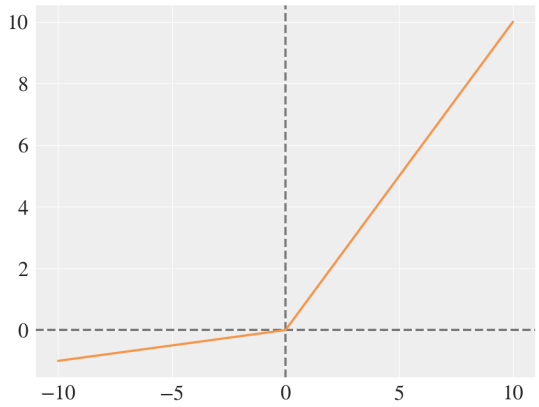
A big novelty has been brought in the field by the introduction of the *Rectified Linear Unit (ReLU)* activation function, defined as:



$$\varphi(x) = \text{ReLU}(x) \doteq \max(0, x) \tag{2.9}$$

Figure 2.8: ReLU activation function.

which is a non-linear version of the linear activation function but, despite the appearances, has a derivative function and allows the use of backpropagation. The main problem of this function is the *dying ReLU problem* instead. In fact, when inputs approach zero, or are negative, the gradient of the function becomes zero and the network can no longer perform backpropagation and learn. The function has therefore several variations and has undergone some improvements, the most important of which is the *Leaky ReLU*:

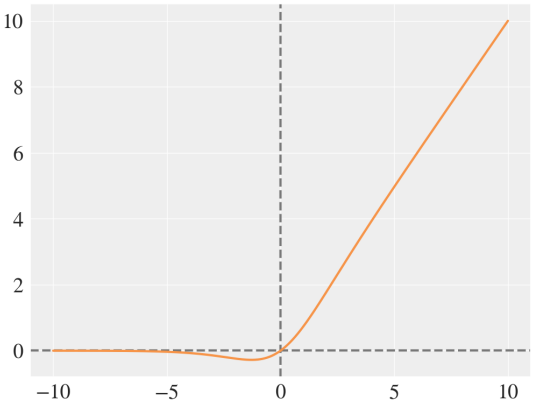


$$\begin{aligned} \varphi(x) &= \text{LeakyReLU}(x) \\ \text{LeakyReLU}(x) &\doteq \max(\alpha \cdot x, x) \end{aligned} \tag{2.10}$$

Figure 2.9: Leaky ReLU activation function.

where α is usually a small number in $[0.1, 0.01]$. This simple variation prevents the problem of the gradient vanishing, enabling backpropagation. Despite that, it may not provide consistent results for negative values since the slope is very small.

A last interesting variation is the *Swish* activation function, defined in [Ramachandran et al., 2017] as:



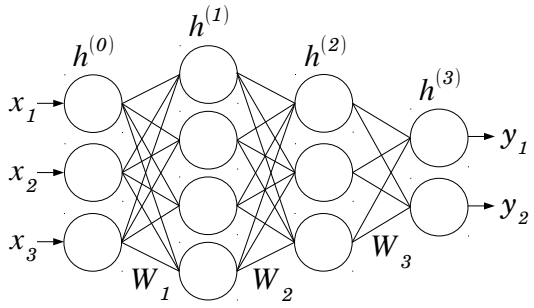
$$\varphi(x) = \text{swish}(x) \doteq x \cdot \sigma(x) \qquad (2.11)$$

Figure 2.10: Swish activation function.

which smoothes the output more than the LeakyReLU and, when the input becomes increasingly negative, it also gets smaller, avoiding large regularization effects.

2.3 Multilayer Perceptron

By composing several single perceptrons together it is possible to create a more complex behaviour, embodied in the model which is called *multilayer perceptron* (MLP). This is often organized in a chain of groups of perceptrons, called layers, with each being a function of the one that preceeds it:



$$\boldsymbol{h}^{(i)} = \varphi^{(i)}\left(\boldsymbol{W}^{(i)T}\boldsymbol{h}^{(i-1)}\right) \qquad (2.12)$$

Figure 2.11: Multilayer perceptron.

where $\boldsymbol{W}^{(i)}$ is the weight matrix and $\varphi^{(i)}$ is the activation function of the i-th layer, and of course $\boldsymbol{h}^{(0)}$ is simply the input vector $\boldsymbol{x}$.

2.3.1 Architectural Considerations

In these chain-based architectures, the main architectural considerations are to choose the depth of the network and the width of each layer. Deeper networks are able to use fewer units per layer and fewer parameters in order to generalize to the test set, but are also often harder to optimize. The *universal approximation theorem* states that an arbitrarily large multilayer perceptron is able to represent an arbitrary function, but it is not guaranteed that the training algorithm is able to learn to represent it. Even if the MLP is able to represent the function, learning can fail for two different reasons. First, the optimization algorithm used for training may not be able to find the value of the parameters that corresponds to the desired function. Second, the training algorithm might choose the wrong function due to overfitting.

The universal approximation theorem also says that there exists a network large enough to achieve any degree of accuracy desired, but the theorem does not say how large this network has to be. Some bounds on the size of a single-layer network needed to approximate a broad class of functions are provided in [Barron, 1993] but, in the worst case, an exponential number of units may be required. To sum up, a feedforward network with a single layer is enough to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly. Therefore, is often preferable to use deeper models which can reduce the number of units required to represent the desired function and can the amount of generalization error.

2.3.2 The Backpropagation Algorithm

In a deep feedforward neural network, such as the MLP represented in Figure 2.11, information flows from the input $\boldsymbol{x}$ to the output $\boldsymbol{y}$. This kind of propagation, where the information is provided from the input layer up to the output layer through the hidden ones, is called *forward propagation*. During the training of the network, this process produces the scalar cost $L(\boldsymbol{x}, \boldsymbol{\theta})$, which depends both on the input vector $\boldsymbol{x}$ and on all the model paramters $\boldsymbol{\theta}$. This function called the *cost function*, or the *loss* of the network, represents how much the neural network is wrong about an output.

The backpropagation algorithm [Rumelhart et al., 1986], allows the information from this loss to flow backwards through the network, in order to compute the gradient of the function. It is important to say that the term backpropagation refers only to the method for computing the gradient, while another algorithm, such as minibatch stochas-

tic gradient descent explained in section 1.2, is actually used to perform the learning. Furthermore, backpropagation is often misunderstood as being specific to multilayer neural networks, but in principle it could compute derivatives of any function.

Computational Graphs and Chain Rule

To understand the backpropagation algorithm, is firstly necessary to introduce the concept of *computational graph*. In a computational graph each variable, which may be a scalar, a vector, a matrix, or a tensor, is represented by a *node*. To represent an operation between variables, these are linked to a new node which returns a single output variable. As an example, the expression $\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}\}$ is reported in Figure 2.12 as a computational graph, which computes a matrix of rectified linear activations $\boldsymbol{H}$ given a minibatch of inputs $\boldsymbol{X}$.

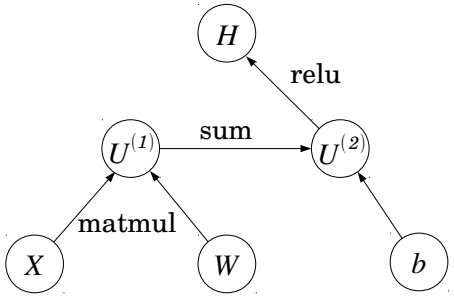


Figure 2.12: Example of a computational graph.

As will be seen shortly, backpropagation is an algorithm that makes great use of the *chain rules* of calculus. This rule is used to compute derivatives of functions formed by composing other functions whose derivatives are known. Generalized in the multivariable case, if $\boldsymbol{y} = g(\boldsymbol{x}) : \mathbb{R}^m \to \mathbb{R}^n$ and $z = f(\boldsymbol{y}) : \mathbb{R}^n \to \mathbb{R}$ then the partial partial derivatives of z with respect to $\boldsymbol{x}$ are expressed as:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i} \qquad \text{or, in vector notation} \qquad \nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z \tag{2.13}$$

From this relation it is possible to see that the gradient of variable $\boldsymbol{x}$ can be obtained by multiplying the Jacobian matrix $\left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top}$ by the gradient $\nabla_{\boldsymbol{y}} z$. The backpropagation algorithm consists exactly in performing such a product for each operation in the graph.

The Algorithm

Using the chain rule, it is immediate to write the algebraic expression for the gradient of a scalar with respect to any node in the computational graph associated.

Consider a computational graph $\mathcal{G}$ describing how to compute a single scalar node $u^{(n)}$, which is obtained starting from the n_i input nodes $u^{(1)} \dots u^{(n_i)}$. Computing the gradients then means to compute:

$$\frac{\partial u^{(n)}}{\partial u^{(i)}} \quad \text{for } i \in \{1, 2, \dots, n_i\} \tag{2.14}$$

In the case of a feedforward neural network, each node of the graph is ordered in such a way that it is possible to compute the output one after the other, starting from $u^{(n_1+1)}$ and ending with $u^{(n)}$. Each node $u^{(i)}$ is also associated with an operation $f^{(i)}$, computed evaluating the function:

$$u^{(i)} = f^{(i)}(\mathbb{A}^{(i)}) \tag{2.15}$$

where $\mathbb{A}^{(i)}$ is the set of all nodes that are connected to $u^{(i)}$.

In order to perform the backpropagation, a computational graph is constructed from $\mathcal{G}$ and adding an extra set of nodes. These form a subgraph $\mathcal{B}$ with one node per node of $\mathcal{G}$. Computation in $\mathcal{B}$ proceeds in the reverse order of computation in $\mathcal{G}$, and each node computes the derivative $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ associated with the forward graph node $u^{(i)}$. This is done using the chain rule with respect to the scalar output $u^{(n)}$:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:u^{(j)} \in \mathbb{A}^{(i)}} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}} \tag{2.16}$$

The subgraph $\mathcal{B}$ has exactly one edge for each edge from node $u^{(j)}$ to node $u^{(i)}$ of $\mathcal{G}$, which is associated with the computation of $\frac{\partial u^{(i)}}{\partial u^{(j)}}$. In addition, a dot product is performed at each node, between the gradient already computed with respect to nodes $u^{(i)}$ that are children of $u^{(i)}$, and the vector containing the partial derivatives $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ for the same children nodes $u^{(i)}$. In the end, the amount of computation required for backpropagation, scales linearly with the number of edges in graph original $\mathcal{G}$, where the computation for each edge corresponds to computing a partial derivative (as well as performing one multiplication and one addition).

The complete back-propagation algorithm is in fact composed of two parts. Firstly, there is need to compute all the activations for each node in the graph, in order to obtain

the final scalar result of node $u^{(n)}$. Secondly, the actual partial derivatives edges are computed, multiplied and stored into an array for the future update of the parameters. The pseudocode for both the two part is reported in Algorithm 3.

Algorithm 3: Forward Activation and Backpropagation Algorithm

Inputs: an input vector $\boldsymbol{x}$

Parameters: a computational graph $\mathcal{G}$ with nodes $u^{(i)}$, the array `grads` that stores the derivatives

Initialization: input nodes $u^{(i)} = x_i$ for $i \in \{1, 2, \ldots, n_i\}$, $\texttt{grads}[u^{(n)}] = 1$

Forward Activation

for i **in** $[n_i + 1, \ldots, n]$ **do**

  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$

Backpropagation

for j **in** $[n - 1, \ldots, 1]$ **do**

  $\texttt{grads}[u^{(j)}] \leftarrow \sum_{i:u^{(j)} \in \mathbb{A}^{(i)}} \texttt{grads}[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$

return $u^{(i)}$, `grads`

2.4 Convolutional Neural Networks

A slightly different type of neural networks, which deserve to be mentioned here, are convolutional neural networks (CNNs). These are specialized in processing data that has a known grid-like topology, such as time-series data (1D) or images (2D).

Differently from MLPs, which are fully connected, CNNs take advantage of the hierarchical pattern in data and assemble more complex patterns using smaller and simpler patterns. In fact these networks employ a mathematical operation called *convolution* between layers instead of the traditional matrix multiplication, which makes them more sparse and more regularized. This also means that CNNs are independent from the input size (unlike MLPs) and allowing them to be trained on data of different sizes. Other than that, CNNs also usually perform another kind of operation, called *pooling*, which both improves the computational efficiency of the network and its ability to generalize.

2.4.1 The Convolution Operation

From a mathematical point of view the convolution operation is a sort of weighted sum, where a function $x(t)$ is averaged with respect to a weight function $w(a)$. The definition in the continuous space is:

$$s(t) = (x * w)(t) \doteq \int x(a)w(t-a)\,\mathrm{d}a \tag{2.17}$$

In the deep learning terminology, the first argument (the function $x(t)$) is often called the *input*, whereas the second argument (the weight function $w(a)$) is known as the *kernel*.

However, usually, both the input and the kernel are discretized, as used for computation, and hence there is need to define its discrete counterpart as:

$$s(t) = (x * w)(t) \doteq \sum_{a=-\infty}^{+\infty} x(a)w(t-a) \tag{2.18}$$

where the variable t can assume only integer values. In deep learning applications, the input is usually a multidimensional array of data and the kernel is usually a multidimensional array of parameters that will be updated by the learning algorithm. Because each element of the input and the kernel must be stored in memory, it is assumed that these functions are zero everywhere but the finite set of points for which the values are stored. This means that in practice the infinite summation is implemented as a summation over a finite number of array elements:

$$s(t) = (x * w)(t) \doteq \sum_{a=-N}^{+N} x(a)w(t-a) \tag{2.19}$$

It is also possible to use convolutions over more than one axis at a time. For example, if we have a two-dimensional image I as our input, we probably also want to use a two-dimensional kernel K:

$$S(i,j) = (I * K)(i,j) \doteq \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.20}$$

A graphical representation of a 2-D convolution is reported in Figure 2.13. It is important to notice that convolution is also commutative, meaning that it is equivalent to write:

$$S(i,j) = (K * I)(i,j) \doteq \sum_m \sum_n I(i-m, j-n)K(m,j) \tag{2.21}$$

The commutative property of convolution arises because the kernel is *flipped* relative to the input, in the sense that as m increases, the index into the input increases, but the index into the kernel decreases. The only reason to flip the kernel is to obtain the commutative property, which is not usually an important property in a neural network implementation. Instead, a more interesting related function is the *cross-correlation*, which is the same as convolution but without flipping the kernel:

$$S(i,j) = (I * K)(i,j) \doteq \sum_m \sum_n I(i+m, j+n) K(m,j) \tag{2.22}$$

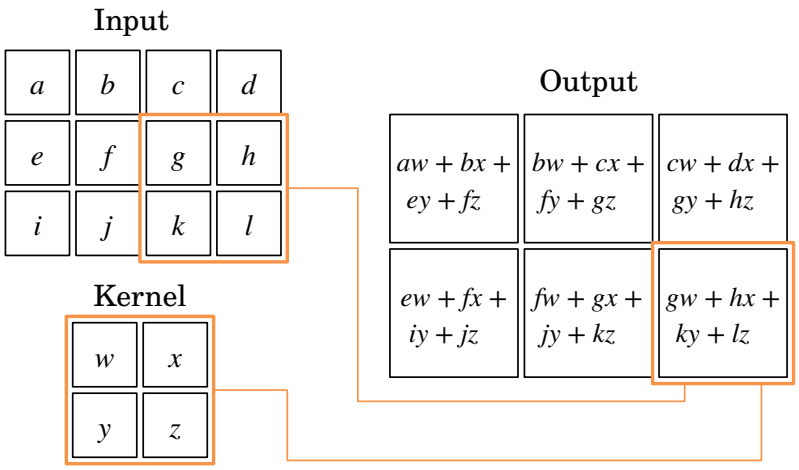


Figure 2.13: 2-D convolution operation.

Convolution Advantages

Convolutional neural networks, thanks to the convolution operation, have therefore three basic properties, which advantage the model during the learning process.

The first one is that convolution leads to *sparse interactions* (or sparse weights): traditional feedforward neural network layers use matrix multiplication to describe the interaction between each input unit and each output unit, which means that every output unit interacts with every input unit. Convolutional neural networks instead have a kernel, wich is smaller than the input, and therefore it also needs fewer parameters. This both reduces the memory requirements of the model and also means that computing the output requires fewer operations.

The second one is that convolution is *sharing parameters*, which refers to using the same parameter for more than one function in the model. In a traditional feedforward neural network, each element of the weights matrix is used exactly once when computing the output of a layer. In a CNN instead, each member of the kernel is used at every position of the input. The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, only one set is learned. This aswell further reduces the storage requirements of the model.

The third one is that the particular form of CNNs parameter sharing causes the layer to have a property called *equivariance* to translation. A function is equivariant if, when the inputs changes, it changes in the same way. Specifically, a function $f(x)$ is equivariant to a function $g(x)$ if:

$$f(g(x)) = g(f(x)) \tag{2.23}$$

When processing time series data, this means that convolution produces a sort of timeline that shows when different features appear in the input. If we move an event later in time in the input, the exact same representation of it will appear in the output, just later in time. Similarly with images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output. By the way, it's important to notice that convolution is not naturally equivariant to some other transformations, such as changes in the scale or rotation of an image. Other mechanisms are necessary for handling these kinds of transformations.

2.4.2 The Pooling Operation

A typical layer of a convolutional network consists of three stages unlike the two of a traditional feedforward network. In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, each linear activation is run through a nonlinear activation function. These two stages have a corresponding version in classical feedforward networks.

The third stage, which has not a corresponding one in the classical form, is composed of a *pooling function* to further modify the output of the layer. A pooling function replaces the output at a certain location with a summary statistic of the nearby outputs. For example, the *max pooling* operation reports the maximum output within a certain

neighborhood, the *average pooling* function include the average of a neighborhood and so on. In all such cases, pooling helps making the representation approximately invariant to small translations of the input, which is a useful property since in many cases it's more important to know whether some feature is present or not than exactly where it is.

Therefore it can be said that the pooling operation summarizes the responses over a whole neighborhood, and it is possible to use fewer units than detector units. This improves the computational efficiency of the network because the next layer has fewer inputs to process. When the number of parameters in the layer is a function of its input size (for example in matrix multiplication) this reduction in the size can also improve the statistical efficiency and reduce memory requirements for storing the parameters.

Pooling is also essential for many different tasks, since it allows for handling inputs of varying size. For example, in a classification problem there might be images of variable size, but the input to the classification layer must have a fixed size. This is accomplished by varying the size of an offset between pooling regions, so that the classification layer always receives the same number of summary statistics regardless of the input size.

Part II

Reinforcement Learning

Chapter 3

Introduction to Reinforcement Learning

3.1 Introduction

Let us leave aside the deep learning framework for a moment, and instead focus on the learning process itself. If we think, for example, of a living being who learns to walk, it will certainly encounter a lot of difficulties and unsuccessful attempts, before actually succeeding in doing it. If, however, he has a clear idea of his intent, every time he will do better than the previous ones and will be encouraged to do even better. On this basis the *reinforcement learning* framework is built, which is different from any other machine learning processes already mentioned.

3.1.1 Differences with (Un)Supervised Learning

Reinforcement learning means learning what to do in order to maximize a certain profit. The learner does not know the best actions to perform a priori, but must instead discover them through a *trial-and-error search* and a *delayed reward* (two of the most distinguish features of reinforcement learning). As will be formalized later, the general idea of reinforcement learning is to recognize the peculiarities of the problem the learning agent is facing, by interacting over time with the environment to reach a goal.

This approach is quite different from *supervised learning*, in which the agent learns from a set of examples labeled by an omniscent source. The purpose of this type of learning is to generalize the response, in order to be able to solve situations not present

in the training set. This is not a solid basis for developing a type of learning based on interaction, since it would be too impractical to obtain examples of desired behaviour that are representative of all the possible situations that can occur to the agent.

Reinforcement learning also differs from *unsupervised learning*, which typically involves learning patterns from a set of unlabeled examples. Although this approach aims to find a structure without knowing the correct behaviour, it does not address the problem of maximizing a reward signal, typical of reinforcement learing problems. Therefore it is possible to consider reinforcement learning as a third machine learning paradigm, alongside supervised and unsupervised learning.

3.1.2 Elements of Reinforcement Learning

Apart from the *agent* that interacts with the *environment*, it is possible to define other elements that constitute the reinforcement learning framework: a *policy*, a *reward signal*, a *value function*, and, in some cases, a *model* of the environment.

The *policy* defines how the agent behaves, according to his state, at a certain moment. Roughly speaking it defines, among all the possible actions, the most profitable action that the agent can take given his state. In some plain cases, the policy can simply consist in a simple function or a lookup table, whereas in other, more difficult tasks, it can be formulated as a more complex predictor, such as a neural network. The policy is certainly one of the fundamental parts of reinforcement learning, since it, even alone, can determine the behavior of the entire agent. It is also important to mention that the policy can be *deterministic*, in which each state corresponds to a single action, or *stochastic*, in which a state corresponds to a distribution of probability of actions.

The *reward signal* defines the goal of the task faced by the agent. In fact, whenever he takes an action, the environment sends him back a *reward* represented by a number, which can be either positive or negative, based on whether it did well or not. The sole purpose of the agent is to maximize the total reward obtainable in the long term.

If the reward signal indicates which action is best to perform in the short term, the *value function* defines what is best in the long run. Practically speaking, the *value* of each state depends on the amount of rewards that the agent can expect to obtain starting from that state. It is obviously possible that a state leads to a generally low reward at the next step, but has an intrinsically high value, as it is followed by states that lead to higher rewards.

The last element, which occurs only in some some reinforcement learning cases, is the *model* of the environment. The model is something that mimics the behavior of the environment, or more generally, that allows us to make predictions about how the environment will behave. Methods for solving reinforcement learning problems that use models are called *model-based* methods, as opposed to simpler *model-free* methods that are explicitly trial-and-error.

It is also important to say that reinforcement learning relies heavily on the concept of *state*, as input to the policy and the value function, and as both input and output from the model. Roughly speaking, the state conveys information to the agent about how the environment is, at a particular time. A proper and rigorous definition of what a system's state is will be given in Section 3.2.

In Figure 3.1 is possible to observe all these elements combined together. The agent is initially in state S_t and, consulting the policy, decides to take action A_t. The environment, in response to that, returns a new state value S_{t+1} and a reward R_{t+1}.

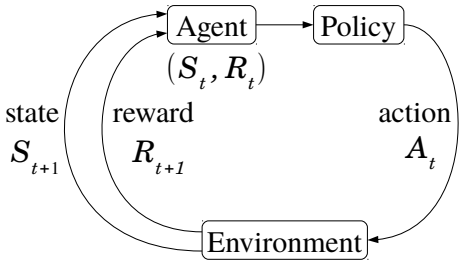


Figure 3.1: Reinforcement learning framework.

The Trade-off between Exploration and Exploitation

One of the most characteristic difficulties to face in reinforcement learning problems is the trade-off between exploration and exploitation. In order to maximize its own reward signal, the agent should take actions that he is already sure of the result, i.e. those actions taken previously and found to be effective in producing reward. But, to do this, he must also find out which of all the possible actions are the most profitable.

The agent has therefore to *exploit* what he already knows in order to obtain the best reward, but it also has to *explore* in order to be able to make better decisions in the future. The difficulty lies in the fact that the agent cannot only pursue a single approach

without failing its task. It must use a combination of the two by trying a variety of actions and progressively favor those that appear the best.

The simplest action selection rule is to select one of the actions that have the highest estimated future reward, such behaviour is called *greedy*. But, as just mentioned, this is not always the way to deal with the problem as we could lose sight of the actions that guarantee us even better rewards. A simple alternative is to behave greedily most of the times, but every once in a while, with small probability ε, instead select randomly from all the actions with equal probability. This near-greedy action selection rule is called *ε-greedy* method and they often performs better than the simple greedy ones because it continue to explore and to improve their chances of recognizing the optimal action.

3.2 Finite Markov Decision Processes

Markov decision processes (MDPs) are a tool which can allow us describe the reinforcement learning problem in a mathematical way, by making rigorous theoretical statements. MDPs, as well as reinforcement learning, involve evaluating a feedback, have an associative aspect (choosing different actions in different situations) and the need to find a tradeoff between immediate and delayed reward and are, therefore, the optimal tool to use. In order to understand how MDPs work however, it is important to start with more basic concepts.

3.2.1 Markov Processes

As written in Section 3.1.2, the agent and the environment interact alternately, the former selecting actions and the latter responding to them presenting new situations and rewards (see Figure 3.1). More specifically, the agent and environment interact at each of a finite sequence of discrete time steps $t = 0, 1, \ldots, T$. At each time step, the agent receives some representation of the environment's state $S_t \in \mathcal{S}$, and therefore it is possible to identify a *trajectory* of states:

$$S_0, S_1, S_2, S_3, \ldots \tag{3.1}$$

This trajectory can be modelized as a *Markov process* (MP), or more precisely a Markov chain since the time is discrete, which is a stochastic model that describes a

sequence of possible events. This process is mathematically defined as a tuple of states-probabilities $(\mathcal{S}, \mathcal{P})$ and has the particular property of being memoryless. This property, also known as the *Markov property* (from which the process takes its name), means that the probability of each possible values for S_{t+1} depends only on the immediately preceding state S_{t-1}, or:

$$\mathcal{P}[S_{t+1}|S_0, \ldots, S_t] = \mathcal{P}[S_{t+1}|S_t] \tag{3.2}$$

This means that each state captures all the relevant information from the process history and, once the state is known, the future is independent of the past states.

In case of a finite MP, the sets of all possible state $\mathcal{S}$ has a finite number n of elements, and it is then possible to define a *transition matrix* as:

$$\mathbf{P} \doteq \begin{bmatrix} p(S_0|S_0) & \ldots & p(S_n|S_0) \\ \vdots & \ddots & \vdots \\ p(S_0|S_n) & \ldots & p(S_n|S_n) \end{bmatrix} \tag{3.3}$$

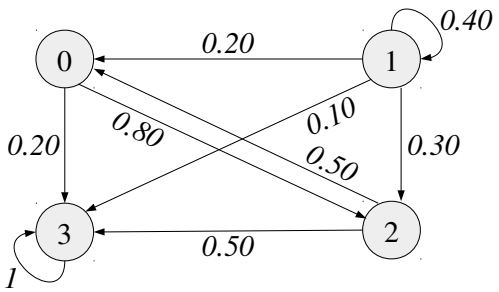
where $p(s'|s)$ is defined as the probability of the transition:

$$p(s'|s) \doteq \mathcal{P}[S_{t+1} = s'|S_t = s] \tag{3.4}$$

It is then straightforward to obtain the property:

$$\sum_{s' \in \mathcal{S}} p(s'|s) = 1 \text{ for all } s \in \mathcal{S}, \tag{3.5}$$

which corresponds to sum over each row in the matrix $\mathbf{P}$. In Figure 3.2 is reported an example of a Markov process with four states, and its corresponing transition matrix.



$$\mathbf{P} = \begin{bmatrix} 0 & 0 & 0.8 & 0.2 \\ 0.20 & 0.40 & 0.30 & 0.10 \\ 0.50 & 0 & 0 & 0.50 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 3.2: Example of a finite Markov process.

3.2.2 Rewards and Return

Moving on to more complicated concepts, we can define a *Markov reward process* (MRP) as a simple Markow process with which a reward functions is associated, in order to evaluate the transition between different states. This new process is mathematically defined as a tuple of state-probabilities-rewards $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{R}$ is the *reward function* and $\gamma \in [0, 1]$ is known as the corresponding *discount factor*. It is then possible to define the *return* G_t as the total discounted reward from a certain time-step t as:

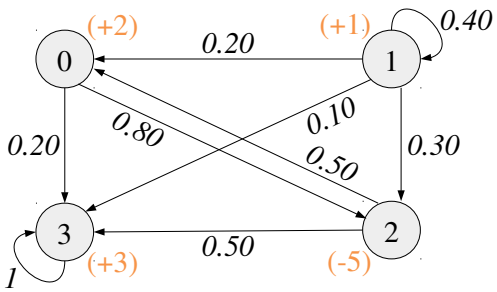
$$G_t \doteq R_{t+1} + \gamma R_{t+1} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.6}$$

This definition allows us to make assessments on how important it is to obtain a reward delayed in time. For example small vales of γ leads to a short-term evaluation, bigger values instead leads to a more far-sighted evaluation. This is also done to avoid infinite returns in cyclic Markov processes.

Similar to how the transition matrix is defined, it is possible to define also the *reward matrix* $\mathbf{R}$, which associate a reward to each state, as:

$$\mathbf{R} \doteq \begin{bmatrix} \mathcal{R}[S_0] \\ \vdots \\ \mathcal{R}[S_n] \end{bmatrix} \tag{3.7}$$

In Figure 3.3 it's possible to see the extension of the example in Figure 3.2, as a Markov reward process with its corresponing reward matrix.



$$\mathbf{R} = \begin{bmatrix} +2 \\ +1 \\ -5 \\ +3 \end{bmatrix}$$

Figure 3.3: Example of a finite Markov process with rewards.

3.2.3 Value Function and Bellman Equation

The next step is trying to evaluate how good a state is, in a generic Markov reward process. In order to do, we can define the *state-value function* as the expected return starting from state s:

$$v(s) \doteq \mathbb{E}[G_t|S_t = s] \tag{3.8}$$

which depends obviously on the parameter γ. It's important to notice that it is possible to decompose the equation in two parts, the immediate reward R_{t+1} and the discounted value of successor state $\gamma v(S_{t+1})$:

$$\begin{aligned} v(s) &\doteq \mathbb{E}[G_t|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} + \dots |S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+2} + \dots)|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1})|S_t = s] \end{aligned} \tag{3.9}$$

where the last part is known as the *Bellman equation* for a generic Markov reward process, and is usually written in the form:

$$v(s) = \mathcal{R}[s] + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s') \tag{3.10}$$

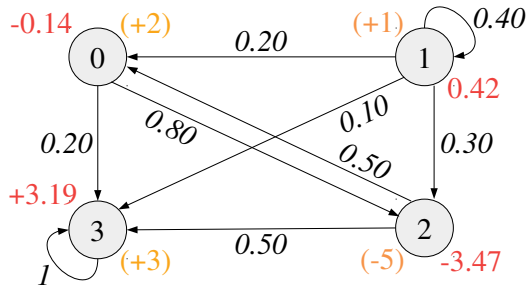
The Bellman equation plays a fundamental role in the study of MRPs because it allows us to calculate the exact value of each state, thus solving our system. It is in fact possible to express the equation using the matrices previously defined:

$$\boldsymbol{v} = \mathbf{R} + \gamma \mathbf{P} v \tag{3.11}$$

which is a linear system, and can be solved directly:

$$\boldsymbol{v} = (\mathbf{I} - \gamma \mathbf{P})^{-1} \mathbf{R} \tag{3.12}$$

However, solving this has a computational complexity of $O(n^3)$ for n states, and is therefore possible only for small MRPs. In Figure 3.4 it is possible to see the values of the states of the example in Figure 3.3, obtained by solving the Bellman equation.



$$\boldsymbol{v} = \begin{bmatrix} -0.14 \\ +0.42 \\ -3.47 \\ +3.19 \end{bmatrix}$$

Figure 3.4: Example of a finite Markov process with values solved.

3.2.4 Actions and Policies

Finally, we can define a *Markov decision process* (MDP) as a Markov reward process, in which an agent can choose a certain action a following a policy π. This new process is mathematically defined as a tuple of states-actions-probabilities-rewards $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{A}$ is a finite set of actions. The *policy* π is defined as the probability distribution over actions given states:

$$\pi(a|s) \doteq \mathcal{P}[A_t = a|S_t = s] \tag{3.13}$$

which fully defines the behaviour of an agent. With this in mind, it is possible to update the definition of the state-value function (3.8) as the expected return starting from state s, and then following policy π:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] \tag{3.14}$$

which brings us to a new form of the Bellman equation:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \quad \text{for all } s \in \mathcal{S}. \end{aligned} \tag{3.15}$$

This equation expresses a relationship between the value of a state and the values of its successor states. A simple representation of this relation can be seen in the diagram in Figure 3.5. Starting from a state s, the root node at the top, the agent could take any

of some set of actions based on its policy π. From each of these, the environment could respond with one of the several next states s', along with a reward r, depending on its dynamics given by the function p.

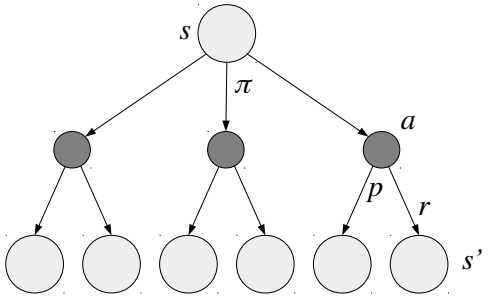


Figure 3.5: Bellman equation diagram.

It is now possible to define the *action-value function*, in analogy with (3.14). This function is defined as the value of taking action a in state s under a policy π, denoted by $q_\pi(s, a)$, as the expected return starting from s, taking the action a, and thereafter following the agent policy π:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \tag{3.16}$$

Optimal Policies and Optimal Value Functions

For finite MDPs, the one in which the state space $\mathcal{S}$ has a finite number of states s, it is possible to precisely define an optimal policy, by taking into account that value functions define a partial ordering over policies. A policy π is in fact said to be better than, or equal, to a policy π', if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. Therefore there is always at least one policy that is better than or equal to all other policies, that is the *optimal policy* and is usually denoted as π_*. This policy is determined by the *optimal state-value function*, denoted with v_* and defined as:

$$v_*(s) \doteq \max_\pi v_\pi(s) \quad \text{for all } s \in \mathcal{S}. \tag{3.17}$$

Optimal policies also share the same *optimal action-value function* q_*, defined as:

$$q_*(s, a) \doteq \max_\pi q_\pi(s, a) \quad \text{for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}. \tag{3.18}$$

For the state-action pair (s, a), this function gives the expected return for taking action a in state s and therefore following the optimal policy. Therefore, we can write q_* in terms of v_* as follows:

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a] \tag{3.19}$$

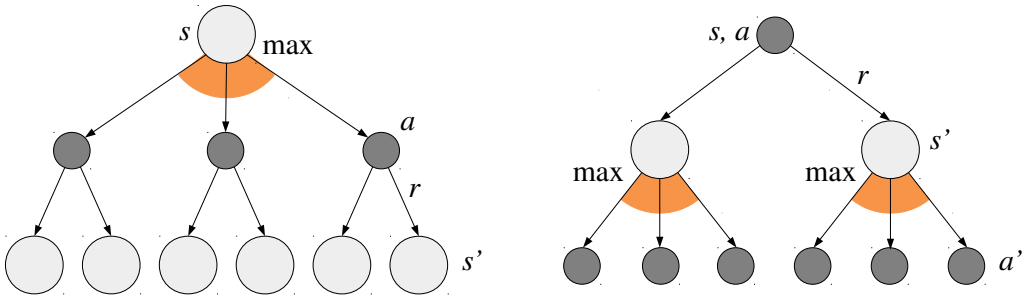
Because v_* is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation in (3.15) but, since it is also the optimal value function, it can be written in a special form without reference to any specific policy, which is the *Bellman optimality equation* for v_*. Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t|S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \end{aligned} \tag{3.20}$$

The Bellman optimality equations for q_* instead is:

$$\begin{aligned} q_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a)|S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a)] \end{aligned} \tag{3.21}$$

The diagrams in Figure 3.6 show graphically the spans of future states and actions considered in the Bellman optimality equation for v_* and q_*. Arcs are depicted at agent's choice points to represent that the maximum over that choice is taken, rather the expected value given some policy.



(a) v_* (b) q_*

Figure 3.6: Bellman optimality equation diagrams.

For finite MDPs, the Bellman optimality equation for v_* has a unique solution. The equation (3.20) is in fact a system of equations, one for each state and, once solved, it is relatively easy to determine an optimal policy. For each state s, there will be one or more actions at which the maximum is obtained in the Bellman optimality equation. Any policy that assign nonzero probability only tho these actions is said to be *greedy* with respect to the optimal evaluation funcition v_*, and is also an optimal policy. The convenience of v_* is in fact that if one uses it to evaluate the short-term consequences of actions, then a greedy policy is actually optimal in the long-term sense, because v_* already takes into account the reward consequences of all possible future behaviour. By means of v_*, the optimal expected long-term return is turned into a quantity that is locally and immediately available for each state. Having q_* makes choosing optimal actions even easier than v_*. For each state s, the agent does not even have to do a one-step-ahead search, but simply has to find any action that maximizes q_*: the action-value function effectively caches the results of all one-step-ahead possible searches.

Chapter 4

Basic Algorithms for Reinforcement Learning

4.1 Dynamic Programming

The term dynamic programming (DP) is both a mathematical and an algorithmic method for solving an optimization problem, by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

In reinforcement learning, it refers to a collection of algorithms that allow us to compute optimal policies, given a perfect model of the environment as a Markov decision process, by estimating and updating the values obtained by the state-value and the action-value functions. Classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and beacuse of their great computational expense, but they are still important theoretically since they provide an essential foundation for the understanding of the other algorithms.

The key idea of dynamic programming, and of reinforcement learning in general, is the use of value functions to organize and structure the search for good policies. It is easy to obtain optimal policies once the optimal value functions for v_* or q_*, expressed in (3.20) and (3.21), are known. In order to do this, DP algorithms aims at continuously evaluating and updating the policy, thanks to the values obtained from the two function. These algorithms are therefore obtained by turning these Bellman equations into update rules for improving approximations of the desired value functions.

4.1.1 Policy Evaluation

We already know how to compute the state-value function v_π for an arbitrary policy:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned}$$

where the existence and uniqueness of v_π is guaranteed as long as either $\gamma < 1$ or eventual termination is guaranteed from all states under policy π. If the environment dynamics are known, then this equation is a system of $|\mathcal{S}|$ simultaneous linear equations and, in principle, its solutions are straightforward computations. However, as already seen, this can be very tedious or computationally expensive and then, also from a generalization perspective, iterative solution methods are more suitable.

Consider a sequence of approximate value functions $v_0, v_1, v_2, \ldots$, each mapping $\mathcal{S}$ to $\mathbb{R}$, where the initial approximation is chosen aribtrarily (except that the terminal state, if any, must be given value 0). Each successive approximation is obtained by using the Bellman equation above as an update rule:

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \quad \text{for all } s \in \mathcal{S}. \end{aligned} \tag{4.1}$$

Clearly, $v_k = v_\pi$ is a fixed point for this update rule because the Bellman equation for v_π assures us of equality in this case. Indeed, it can be shown that the sequence $\{v_k\}$ converges to v_π, as $k \to \infty$ under the same confitions that guarantee the existence of v_π. This algorithm is called *iterative policy evaluation.*

To produce each successive approximation, v_{k+1} from v_k, iterative policy evaluation applies the same operation to each state s: it replaces the old value with a new one obtained from the old values of the successor states of s, and the expected immediate rewards. This kinf of operation is called *expected update* and there are several different kind of them, depending on whether a state or a state-action pair is being updated, and depending on the precise way the estimated values of the successor states are combined.

To write a sequential program to implement the algorithm, one should use two arrays, one for the old values v_k and one for the new ones v_{k+1}. In fact, it is easier to use one array and update the values "in place", thus sometimes new values are used instead of old ones on the right hand side of (4.1). This in-place algorithm also converges to v_π, in

fact it usually converges faster than the two arrays version, because it uses new data as soon as they are available. The pseudocode for the algorithm is given in Algorithm 4.

Algorithm 4: Iterative Policy Evaluation

Inputs: the policy to be evaluated π

Parameters: a small threshold $\theta > 0$ used to determine the accuracy of the estimation

Initialization: $V(s)$, for all $s \in \mathcal{S}$, arbitrarily except $V(final) = 0$

$\Delta \leftarrow \theta' > \theta$

while $\Delta > \theta$ **do**

  $\Delta \leftarrow 0$

  for s **in** $\mathcal{S}$ **do**

    $v \leftarrow V(s)$

    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

return $V \approx v_\pi$

4.1.2 Policy Improvement

Once a way of evaluating a policy is defined, it is important to also find a method to determine if it is better to change the current policy, to deterministically choose an action instead of following the probability distribution $\pi(s)$. We know how good is to follow the current policy from a state s, because it is represented by the value $v_\pi(s)$, but we don't know if it would be better or worse to change to a new policy. To answer that we need to consider selecting action a in s and thereafter following the current policy π. The value of this way of behaving is:

$$\begin{aligned} q_\pi(s, a) &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{4.2}$$

The criterion here is whether this value is greater or less than $v_\pi(s)$. If it is greater, then it is better to select a once in s and therefore follow π rather than to follow π all the

time, and it is possibile to modify the policy in such a way that a is followed every time s is encountered. That this is true is a special case of a general result called the *policy improvement theorem*. Let π and π' be any pair of deterministic policies such that:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \quad \text{for all } s \in \mathcal{S} \tag{4.3}$$

Then the policy π' must be as good as, or better than, the policy π. That is, it must obtain greater or equal expected return from all states $s \in \mathcal{S}$:

$$v_{\pi'}(s) \geq v_\pi(s) \tag{4.4}$$

The policy improvement theorem applies of course in the case of the two former policies: the original deterministic policy π and a changed policy π' which is identical to π except that $\pi'(s) = a \neq \pi(s)$. For states other than s, (4.3) holds because the two sides are equal. Thus, if $q_\pi(s, a) > v_\pi(s)$, then the changed policy is better than π.

The idea behind the proof of this theorem, starting from (4.3), is:

$$\begin{aligned}
v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\
&= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = \pi'(s)] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}[R_{t+2} + \gamma v_\pi(S_{t+2}) | S_{t+1}, A_{t+1} = \pi'(s)] | S_t = s] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) | S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+3}) | S_t = s] \\
&\vdots \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \ldots | S_t = s] \\
&= v_{\pi'}(s)
\end{aligned} \tag{4.5}$$

It is then a natural extension to consider changes in all states and all possible actions, selecting at each state the action that appears to be the best according to $q_\pi(s, a)$. In other words, to consider the new *greedy* policy π', given by:

$$\begin{aligned}
\pi'(s) &\doteq \arg\max_a q_\pi(s, a) \\
&= \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\
&= \arg\max \sum_{s', r} p(s', r | s, a)[r + \gamma v_\pi(s')]
\end{aligned} \tag{4.6}$$

This greedy policy takes the action that looks best in the short term according to v_π and, since it meets the conditions of the policy improvement theorem, it is as good, or better than, the original policy. This overall process of making a new policy and improving the original one, is called *policy improvement*.

4.1.3 Policy Iteration

Once the policy π has been improved using v_π to a better policy π', it is possible to compute $v_{\pi'}$ and improve it again to an even better policy π''. This sequence of improving policies can be represented as:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

where E stands for policy evaluation and I stands for policy improvement. This way of finding an optimal policy is called *policy iteration* and the pseudocode of the corresponding algorithm is given in Algorithm 5.

Algorithm 5: Policy Iteration (using iterative policy evaluation)

Inputs: the policy to be evaluated and improved π
Parameters: a small threshold $\theta > 0$ used to determine the accuracy of the estimation
Initialization: $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

```
policy_stable ← False
while not policy_stable do
    run Algorithm 4: Iterative Policy Evaluation
    policy_stable ← True
    for s in S do
        old_action ← π(s)
        π(s) ← arg max_a Σ_{s',r} p(s', r|s, a)[r + γV(s')]
        if old_action ≠ π(s) then
            policy_stable ← False
return V ≈ v_*, π ≈ π_*
```

Since a finite MDP has a finite number of policies, this process must converge to an optimal policy and an optimal value function in a finite number of iterations. It is immediate to notice that it is formed by the policy evaluation algorithm described in Algorithm 4, and by a loop that simply updates the policy based upon the results.

4.1.4 Value Iteration

The main problem behind policy iteration method is that each of its iterations requires a policy evaluation which consequently requires computation time, since the convergence exactly to v_π occurs only in the limit. Actually, the evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of the algorithm. The most important case is when policy evaluation is stopped after just one iteration (one update of each state). This algorithm is called *value iteration*. It can be written as a particularly simple update rule that combines the policy improvement and truncated policy evaluation steps:

$$\begin{aligned} v_{k+1}(s) &\doteq \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \quad \text{for all } s \in \mathcal{S}, \end{aligned} \tag{4.7}$$

which for arbitrary v_0 can be shown to converge to v_* under the same conditions that guarantee the existence of v_*.

Another way of understanding value iteration is by reference to the Bellman optimality equation (3.20), since it is obtained simply by turning this into an update rule. It may be also noticed how the value iteration update is identical to the policy evaluation update (4.1), except that it requires the maximum to be taken over all actions.

The pseudocode for the corresponding algorithm is reported in Algorithm 6. Like policy evaluation, value iteration formally requires an infinite number of iterations to converge exactly to v_* but, in practice, it's common to stop once the value function changes by only a small amount in the same iteration. This algorithm effectively combine, in each of its iterations, an iteration of policy evaluation and one iteration of policy improvement. Faster convergence is often achieved by interposing multiple policy evaluation iterations between each policy improvement iteration. In general, the entire class of truncated policy iteration algorithms can be thought of as sequences of sweeps, some of which use policy evaluation updates and some of which use value iteration updates.

Because the max operation in (4.7) is the only difference between these updates, this just means that the max operation is added to some sweeps of policy evaluation. All of these algorithms converge to an optimal policy for discounted finite MDPs.

Algorithm 6: Value Iteration

Inputs: the policy to be evaluated and improved π

Parameters: a small threshold $\theta > 0$ used to determine the accuracy of the estimation

Initialization: $V(s)$, for all $s \in \mathcal{S}$, arbitrarily except $V(final) = 0$

$\Delta \leftarrow \theta' > \theta$

while $\Delta > \theta$ **do**

  $\Delta \leftarrow 0$

  for s **in** $\mathcal{S}$ **do**

    $v \leftarrow V(s)$

    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

$\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

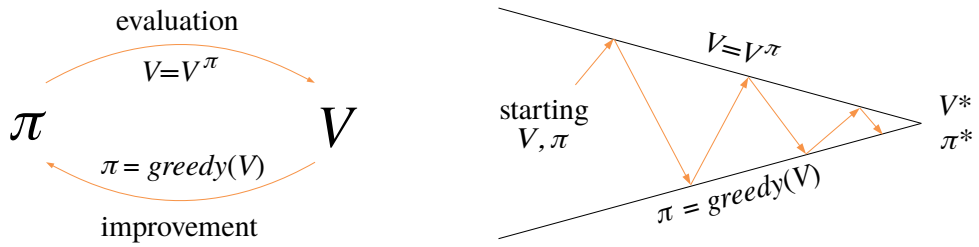
return $\pi \approx \pi_*$

4.1.5 Generalized Policy Iteration

The term *generalized policy iteration* (GPI) is used to refer to the genral idea of letting policy evaluation and policy improvement processes interact. Almost all reinforcement learning methods are easily described as GPI, in fact all have identifiable policies and value functions, with the policy always being improved with respect to the value function and the value function always being driven toward the value function for the policy. If both the evaluation process and the improvement process stabilize, then the policy and the value function must be both the optimal ones.

The evaluation and improvement processes in generalized policy iteration can be viewed as both as competing and cooperating. They compete in the sense that they pull in opposing directions. making the policy greedy with respect to the value function

tipically makes the value function incorrect for the changed policy, and making the value function consistent with the policy tipically causes that policy no longer to be greedy. In the long run, however, these two processes interact to find the optimal value function and the optimal policy. The double evaluation-improvement loop is graphically represented in Figure 4.1. In the diagram the processes are represented as lines, which constrain the iteration and guide it to the convergence with optimal results.



(a) loop (b) diagram

Figure 4.1: Generalized policy iteration loop and diagram.

4.2 Monte Carlo Methods

Monte Carlo (MC) methods are a class of reinforcement learning algorithms wich does not require a complete understanding of the environment but only need experience, a sample sequences of state-action-reward from the interaction between the agent and the environment. Even if a model is required, this only need to generate sample transitions, not the complete probability distributions of all possible transitions that is required for dynamic programming instead.

4.2.1 First-Visit Monte Carlo

The first algorithm presented aims to learn the state-value function for a given policy. As said before, the value of a state is the expected return starting from that state. The most 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.

Suppose we want to estimate $v_\pi(s)$, the value of state s under policy π, given a set of episodes obtained following π and passing through s. Each occurrence of state s in an episode is called a *visit* to s. Actually the same state s could be visited many times during the same episode, so it's important to distinguish the first time s is visited by calling it *first-visit*. The *first-visit Monte Carlo method* estimates $v_\pi(s)$ as the average of the returns following first visit to s and is different from the *every-visit Monte Carlo method* which averages the returns following all visit to s. Both methods converge to $v_\pi(s)$ as the number of visits (or first visits) to s goes to infinity.

Algorithm 7: First-visit Monte Carlo

Inputs: the policy to be evaluated π

Initialization: $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$ and `returns`(s), an empty list for all $s \in \mathcal{S}$

for each episode do

- `episode` $\leftarrow$ new episode following $\pi : S_0, A_0, R_1, S_1, \ldots, S_{T-1}, A_{T-1}, R_T$
- $G \leftarrow 0$
- **for** t **in** $[T-1, T-2, \ldots, 0]$ **do**
  - $G \leftarrow \gamma G + R_{t+1}$
  - **if** S_t **not in** $[S_0, \ldots, S_{t-1}]$ **then**
    - append G to `returns`(S_t)
    - $V(S_t) \leftarrow$ avg(`returns`(S_t))

return $V \approx v_\pi$

It is possible now to make a comparison between this and the dynamic programming approach. In the latter one all of the probabilities must be computed before the algorithms can be applied, and such computations are often complex or error-prone. In contrast, generating the sample required by Monte Carlo method is quite easy. Also, whereas the dynamic programming diagram includes only one-step transitions, the Monte Carlo diagram (see Figure 4.2) goes all way to the end of the episode. Whereas the DP diagrams are represented with all possible interactions(see Figure 3.6), the MC diagram shows only those sampled on current the episode. It is important to notice that the

computational expense of estimating the value of a single state is dependent only on the duration of the episode, and independent from the total number of states.

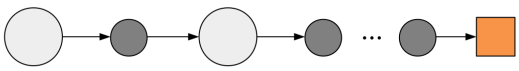


Figure 4.2: Monte Carlo diagram.

4.2.2 Monte Carlo with Exploring Starts

If a model is not available, then it is particularly useful to estimate *action values* (the values of state-action pairs) rather than state values. The policy evaluation problem is then to estimate $q_\pi(s, a)$, the expected return when starting in state s, taking action a, and following policy π. The only complication is that many state-action pairs may never be visited. If π is a deterministic policy, then in following π one will observe returns only for one of the actions from each state. With no returns to average, the Monte Carlo estimates of the other actions will not improve with experience.

This is the already known problem of maintaining exploration, and one way to solve this is to specify that the episode starts in a state-action pair, and that every pair has a nonzero probability of being selected as the start. This guarantess that all the state-action pairs will be visited an infinite number of times in the limit of an infinite number of episodes. This assumption is usually called *exploring starts*.

It is now possible to combine the ideas of MC methods with the GPI explained in 4.1.5. To begin , let us consider a MC version of classical policy iteration. In this method there is an alternating evaluation of the policy followed by an improvement, as the one described in 4.1.3, but this time the action-value function is taken into consideration rather than the state-value:

$$\pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} q_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} q_*$$

Here policy improvement is done in the same way as before, apart from making the policy greedy with respect to the action-value function:

$$\pi(s) \doteq \arg\max_a q(s, a) \tag{4.8}$$

Actually, an unlikely assumption was made to easily obtain the guarantee of convergence for the Monte Carlo method, that is that the policy evaluation could be done with an infinite number of episodes. This is relatively easy to remove, in fact the same issue arises even in DP methods, which also converge only asymptotically to the true value function. In both DP and MC cases there are two ways to solve the problem. One is to hold firm the idea of approximating q_{π_k} in each policy evaluation. This approach can probably be made completely satisfactory in the sense of guaranteeing correct convergence up to some level of approximation. In the second approach, on each evaluation step the value function is moved toward q_{π_k}, but it is not expected to actually get close except over many steps, and the attempt to completely evaluate the policy is given up.

For Monte Carlo policy iteration it is natural to alternate between evaluation and improvement on an episode-by-episode basis. After each episode, the observed returns are used for policy evaluation, and then the policy is improved at all the states visited in the episode. The pseudocode of the algorithm, called *Monte Carlo with Exploring Starts* is reported in Algorithm 8.

Algorithm 8: Monte Carlo with Exploring Starts

Inputs: the policy to be evaluated and updated π
Initialization: $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$ and $\texttt{returns}(s)$, an empty list for all $s \in \mathcal{S}$

for each episode do
  $(S_0, A_0) \leftarrow$ chosen randomly from $\mathcal{S}, \mathcal{A}(S_0)$
  $\texttt{episode} \leftarrow$ new episode following $\pi : S_0, A_0, R_1, S_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  for t **in** $[T-1, T-2, \ldots, 0]$ **do**
    $G \leftarrow \gamma G + R_{t+1}$
    if (S_t, A_t) **not in** $[S_0, A_0 \ldots, S_{t-1}, A_{t-1}]$ **then**
      append G to $\texttt{returns}(S_t)$
      $Q(S_t, A_t) \leftarrow \text{avg}(\texttt{returns}(S_t, A_t))$
      $\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$
return $\pi \approx \pi_*$

In this algorithm, all the returns for each state-action pair are accumulated and averaged, irrespective of what policy was in force when they were observed. It is easy to see that this cannot converge to any suboptimal policy. If it did, then the value function would eventually converge to the value function for that policy, and that in turn would cause the policy to change. Stability is achieved only when both the policy and the value function are optimal.

4.3 Temporal-Difference Learning

Temporal Difference (TD) learning combines both ideas from Monte Carlo and dynamic programming. Like in MC methods, TD techniques can learn from direct experience aswell, without knowing a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting to reach the final outcome of the episode (unlike MC methods).

4.3.1 One-Step Temporal-Difference

As just mentioned, both MC and TD methods use experience to solve the problem of finding the correct estimate V of v_π. But, unlike Monte Carlo methods which have to wait until the reaching of the final state, TD methods only wait until the next time step. So, at time $t+1$, it is immediately possible to make an estimate of the value V, taking into account the observed reward R_{t+1} and the estimate of $V(S_{t+1})$. The simplest TD method makes the update:

$$V(S_{t+1}) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{4.9}$$

where it is easy to note that the target for the MC update is G_t whereas the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$. This method is called *TD(0)* or *one-step TD* and, since the update is based in part upon an existing estimate, it is a *bootstrapping* method. The corresponding pseudocode is reported in Algorithm 9.

One important remark to make is that the term in brackets in the one-step TD update is a sort of error, which measures the difference between the estimated value $V(S_t)$ and the better estimate $R_{t+1} + \gamma V(S_{t+1})$. This quantity is called the *TD error* and appears in various areas of reinforcement learning:

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{4.10}$$

It is important to notice that this error is the estimate made at a specific time, since it depends on the next state and next reward, which are not available until one step later. That is, δ_t is the error for $V(S_t)$ available at time $t+1$.

Algorithm 9: One-Step Temporal-Difference

Inputs: the policy to be evaluated π
Parameters: step size $\alpha \in [0, 1]$
Initialization: $V(s)$, for all $s \in \mathcal{S}$, arbitrarily except for $V(final) = 0$

for each episode do
  $S \leftarrow$ new state
  for step in episode do
    $A \leftarrow$ new action following π
    $R, S' \leftarrow$ observed reward and next state
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
return $v_\pi \approx v_*$

The one-step TD diagram, visible in Figure 4.3 is quite similar to the MC one in Figure 4.2, but with just one step of update. Since the estimate is updated on the basis of a sample transition, this type of updates are usually referred to as *sample updates* which 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.

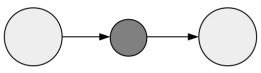


Figure 4.3: Temporal-Difference diagram.

Finally, TD methods have two main advantage over the methods previously explained. TD, unlike DP methods, does not require a model of the environment, nor of the rewards

and next-state probabilities. The other advantage, which makes TD standing out with respect to MC methods is that they does not need to reach the end of the episode but just the next step, and this is often critical since some applications have very long episodes.

4.3.2 SARSA

As done for MC methods, the next step is to apply the idea behind the GPI, but this time using TD methods for the evaluation and the prediction part. As before, this is made by learning the action-value function rather than the state-value function.

It is important now to distinguish between *on-policy* and *off-policy* methods. In the former case, the agent learns and behaves using the same policy while, in the latter, the agent learns and behaves using two (slightly) different versions of the policy. In fact, the SARSA algorithm uses the same policy to both learn and behave and is therefore an on-policy method, while the Q-Learning algorithm, explained in the next section, uses a modification of the behaveviour policy to update it and is therefore an off-policy method.

In the SARSA algorithm transitions from state-action pair to state-action pair are considered, and the corresponding values are learned. The theorem which assures the convergence of the state values under the one-step TD also applies to the correspondig algorithm for action values:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{4.11}$$

This update rule uses every element of the quintuple of events $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that make up a transition from one state-action pair to another, and which give rise to the *SARSA* name of the algorithm. The corresponding diagram is reported in Figure 4.4, which is quite similar to the one-step TD reported in Figure 4.3, but it starts and ends with an action instead of a state.

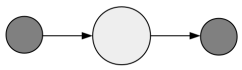


Figure 4.4: SARSA diagram.

The algorithm continually estimates q_π for the behaviour policy π, and at the same time change π in a greedy way with respect to π. SARSA converges to an optimal policy

and action-value function as long as all state-action pairs are visited an infinite number of times. The corresponding pseudocode is reported in Algorithm 10.

Algorithm 10: SARSA (On-Policy Temporal-Difference)

Inputs: the policy to be evaluated and improved π
Parameters: step size $\alpha \in (0, 1]$, small number $\varepsilon > 0$
Initialization: $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}$, arbitrarily except for $Q(final) = 0$

for each episode do
  $S \leftarrow$ new state
  $A \leftarrow$ action from S following policy derived from Q (e.g. ε-greedy)
  for step in episode do
    $R, S' \leftarrow$ observed reward and next state after taking A
    $A' \leftarrow$ action from S' following policy derived from Q (e.g. ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
    $S \leftarrow S'$
    $A \leftarrow A'$
return $Q \approx q_*, \pi \approx \pi_*$

4.3.3 Q-learning and Expected SARSA

Is is now possible to obtain, with a slightly modification of the update rule, the corresponding off-policy method called *Q-learning*:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{4.12}$$

In this case, the learned action-value function Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed, which dramatically simplifies the analysis of the algorithm and enables early convergence proofs. The pseudocode for the algorithm is reported in Algorithm 11.

Algorithm 11: Q-learning (Off-Policy Temporal-Difference)

Inputs: the policy to be evaluated and improved π

Parameters: step size $\alpha \in (0, 1]$, small number $\varepsilon > 0$

Initialization: $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}$, arbitrarily except for $Q(final) = 0$

for each episode do

  $S \leftarrow$ new state

  for step in episode do

    $A \leftarrow$ action from S following policy derived from Q (e.g. ε-greedy)

    $R, S' \leftarrow$ observed reward and next state after taking A

    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

    $S \leftarrow S'$

return $Q \approx q_*, \pi \approx \pi_*$

It is also possible to consider a slightly modification of the Q-learning algorithm, except that instead of the maximum over next state–action pairs it uses the expected value, taking into account how likely each action is under the current policy. The corresponding the update rule therefore is:

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1})|S_{t+1}] - Q(S_t, A_t)] \\ &\leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)] \end{aligned} \tag{4.13}$$

Given the next state S_{t+1}, this algorithm moves *deterministically* in the same direction as SARSA moves but *in expectation*, and accordingly it is called *Expected SARSA*.

Expected SARSA is more computationally complex than SARSA but, in return, it eliminates the variance due to the random selection of A_{t+1}. Given the same amount of experience it usually perfom slightly better than SARSA. Furthermore, despite it is usually used as an on-policy method, in general it might use a different policy from the target policy π to generate behaviour, in which case it becomes an off-policy method. In this sense, Expected SARSA subsumes and generalizes Q-learning while reliably improving over SARSA. Even if the algorithm is easy to deduce from the Q-learning algorithm, the pseudocode is reported for sake of completeness in Algorithm 12.

Algorithm 12: Expected SARSA

Inputs: the policy to be evaluated and improved π

Parameters: step size $\alpha \in (0, 1]$, small number $\varepsilon > 0$

Initialization: $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}$, arbitrarily except for $Q(final) = 0$

for each episode do

  $S \leftarrow$ new state

  for step in episode do

    $A \leftarrow$ action from S following policy derived from Q (e.g. ε-greedy)

    $R, S' \leftarrow$ observed reward and next state after taking A

    $Q(S, A) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)]$

    $S \leftarrow S'$

return $Q \approx q_*, \pi \approx \pi_*$

The corresponding diagrams for these algorithms, reported in Figure 4.5, are quite different from the previous ones, as they take into account all the possible actions for the state of the system. The only difference is the max operation between the following actions, which emphasizes the similarity between the two methods.

(a) Q-learning (b) Expected SARSA

Figure 4.5: Q-learning and Expected SARSA diagrams.

4.4 Summary

All of the methods presented in this chapter have three key ideas in common: first, they all seek to estimate value functions (be they state-value of action-value functions); second, they all operate by updating values along actual or possible state trajectories; and third, they all follow the general strategy of generalized policy iteration (GPI), meaning that they maintain an approximate value function and an approximate policy, and they continually improve each on the basis of the other.

Each one of the algorithms however has a coherent set of ideas cutting across methods. Each idea can be viewes as a dimension along which methods vary. The set of such dimensions spans a large space of possible methods. Two of the most important dimensions along which these methods may vary are shown in Figure 4.6. These dimensions have to do with the kind of update used to improve the value function.

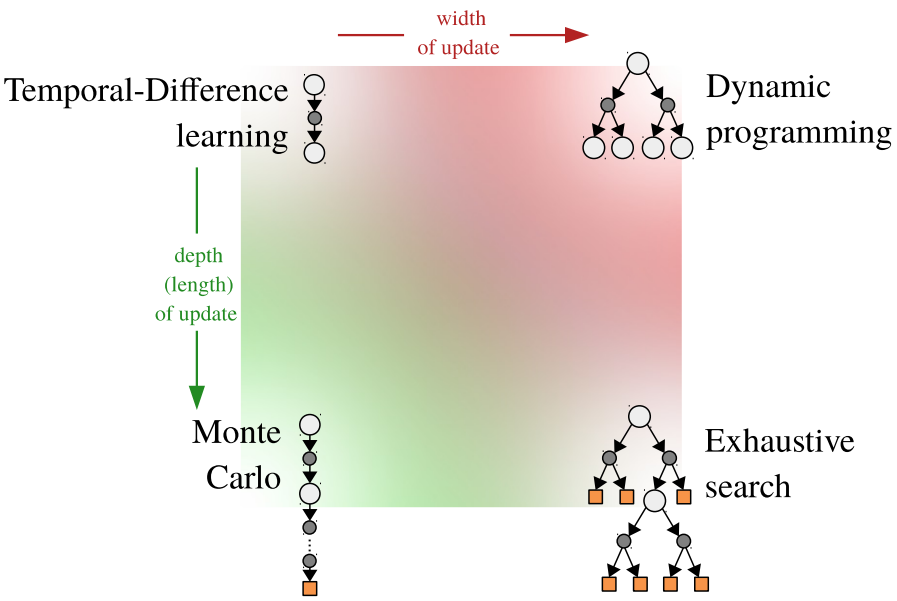


Figure 4.6: Span of the methods space.

The horizontal dimension is whether they are sample updates (based on a sample trajectory) or expected updates (based on a distribution of possible trajectories). Expected

updates require a distribution model, whereas sample updates need only a sample model, or can be done from actual experience with no model at all (another dimension of variation). The vertical dimension of Figure 4.6 corresponds to the depth of updates, that is, to the degree of bootstrapping. At three of the four corners of the space are the three primary methods for estimating values: dynamic programming, Temporal-Difference, and Monte Carlo. Along the left edge of the space are the sample-update methods, ranging from one-step TD updates to full-return MC updates.

Dynamic programming methods are shown in the extreme upper-right corner of the space because they involve one-step expected updates. The lower-right corner is the extreme case of expected updates so deep that they run all the way to terminal states, this is the case of exhaustive search. The interior of the square is filled in to represent the space of all such intermediate methods.

Chapter 5

Deep Reinforcement Learning

5.1 Prediction and Control with Approximation

Trying to apply the reinforcement learning framework to different problems, it is immediate to find tasks that are too difficult to be solved in a decent amount of computational time and with sufficient precision. Many problems arise, for example, when a large state space $\mathcal{S}$ is taken into consideration. Not only solving a problem of this kind, using previously explained algorithms, would require a huge amount of memory, but also of time and data. In many of the target tasks in fact, almost every state would have never been seen before. To make the best action in these states it is necessary for the agent to learn to *generalize* correctly, by analyzing different states that are similar to the current one.

5.1.1 Function Approximation

The kind of generalization used is called *function approximation*, which is an instance of *supervised learning*, because it takes examples from a desired function (e.g. the state-value function) and attempts to generalize from them, in order to construct an approximation of the entire function.

The main difference between such a technique and the previously explained ones, is that the approximate value function is no longer represented by the fixed association state-value, but is a parametrized functional form with a weight vector $\boldsymbol{w} \in \mathbb{R}^d$. For example, the approximate value function $\hat{v}(s, \boldsymbol{w})$ might now be a linear function of features of the state, with $\boldsymbol{w}$ vector of weights. More generally, $\hat{v}$ might be the function computed by a deep artificial neural network, such as the ones described in Chapter 2,

with $\boldsymbol{w}$ the vector of connection weights between the hidden layers units.

Typically, the number of weigths is much less than the number of states $d \ll |\mathcal{S}|$ and, therefore, changing one weight changes the estimated value of many states. As a consequence of this, when a single state is updated, the change generalizes from that state to affect the values of many other states.

5.1.2 The Prediction Objective $\overline{VE}$

In previously described algorithms, an explicit objective to minimize was not necessary since the learned value function would have converged, sooner or later, to the true value function exactly. Moreover, the learned values at each state were decoupled, an update at one state affected no other. But, with function approximation, an update affects all states, and it is not possible to get the values of all states exactly correct. By assumption we have far more states than weights, so making one state's estimate more accurate invariably means making others less accurate. Therefore, there is need to specify a state distribution $\mu(s) \geq 0$, with $\sum_s \mu(s) = 1$, representing how much the error in a state s is more important than the errors in other states.

The error in a general state s is defined as the mean square of the difference between the approximate value $\hat{v}(s, \boldsymbol{w})$ and the true value function $v_\pi(s)$. By weigthing this over the distribution $\mu(s)$ it is possible to define the *Mean Squared Value Error*:

$$\overline{VE}(\boldsymbol{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) \left[\hat{v}(s, \boldsymbol{w}) - v_\pi(s)\right]^2 \tag{5.1}$$

The square root of $\overline{VE}$ gives a rough measure of how much the approximate values differ from the true values. Often $\mu(s)$ is chosen to be the fraction of time spent in s.

An ideal goal in terms of $\overline{VE}$ would be to find a *global minimum*, which is represented by a weight vector $\boldsymbol{w}^*$ for which $\overline{VE}(\boldsymbol{w}^*) \leq \overline{VE}(\boldsymbol{w})$ for all possible $\boldsymbol{w}$. In practical terms this is achieved quite easily by linear approximators, but is sometimes very difficult for non-linear ones, which instead converge more often to a *local minimum*, which is represented by a weight vector $\boldsymbol{w}^*$ for which the relation $\overline{VE}(\boldsymbol{w}^*) \leq \overline{VE}(\boldsymbol{w})$ holds only in a neighborhood of $\boldsymbol{w}^*$. Said that, it is always possible that the local minimum for the non-linear approximator is at a lower value than the global minimum for the linear approximator, and therefore it still represents a better solution.

5.2 Stochastic-gradient and Semi-gradient Methods

We now present a class of learning methods for function approximation in value prediction, based on the stochastic gradient descent described in 1.2.1. In these methods the weight vector is a vector with a fixed number of components $\boldsymbol{w} = (w_1, w_2, \ldots, w_d)$, and the approximate value function $\hat{v}(s, \boldsymbol{w})$ is differentiable with respect to $\boldsymbol{w}$ for each state $s \in \mathcal{S}$. The weight vector will be updated at each of a series of discrete time steps $t = 0, 1, 2, \ldots$, so there is need to a notation $\boldsymbol{w}_t$ for the weight vector at each step. At each step, a new example $S_t \to v_\pi(S_t)$ is observed, consisting of a state S_t and its true value under the policy. These states might be successive states from an interaction with the environement, but this is not restrictive.

Even with the correct values $v_\pi(S_t)$, this is still a difficult problem. Since the function approximator has limited resources (the fixed lenght weight vector $\boldsymbol{w}$), there is generally no $\boldsymbol{w}$ that gets all the states exactly correct and, in addition, we need to generalize to all the states not appeared in the examples. A good strategy in this case, is to minimize the error $\overline{VE}$ in the observed examples using the gradient descent. In order to do so, the weight vector is adjusted by a small amount in the direction that would most reduce the error on that example:

$$\begin{aligned} \boldsymbol{w}_{t+1} &\doteq \boldsymbol{w} - \frac{1}{2}\alpha \nabla_{\boldsymbol{w}} \left[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}_t)\right]^2 \\ &= \boldsymbol{w} + \alpha \left[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}_t)\right] \nabla_{\boldsymbol{w}} \hat{v}(S_t, \boldsymbol{w}_t) \end{aligned} \tag{5.2}$$

where α is a positive step-size parameter.

5.2.1 Gradient Monte Carlo

We now study the case in which the target output is not the true value $v_\pi(S_t)$, but some approximation U_t, it might be a noise-corrupted version or a bootstrapped estimation. This yelds to the general stochastic gradient descent method:

$$\boldsymbol{w}_{t+1} \doteq \boldsymbol{w} + \alpha \left[U_t - \hat{v}(S_t, \boldsymbol{w}_t)\right] \nabla_{\boldsymbol{w}} \hat{v}(S_t, \boldsymbol{w}_t) \tag{5.3}$$

In the particular case U_t is unbiased, that is:

$$\mathbb{E}[U_t | S_t = s] = v_\pi(S_t)$$

then, $\boldsymbol{w}_t$ is guaranteed to converge to a local minimum.

Because the true value of a state is the expected value of the return following it, the Monte Carlo target $U_t = G_t$ is by definition an unbiased estimate of $v_\pi(S_t)$. With this choice, the general stochastic gradient descent method converges to a locally optimal approximation of $v_\pi(S_t)$. Thus, the gradient descent version of Monte Carlo state-value prediction is guaranteed to find a locally optimal solution. The pseudocode for this algorithm is shown in Algorithm 13.

Algorithm 13: Gradient Monte Carlo

Inputs: the policy to be evaluated π, a differentiable function $\hat{v}$
Parameters: step size $\alpha > 0$
Initialization: weight vector $\boldsymbol{w}$ arbitrarily

for each episode do
    `episode` $\leftarrow$ new episode following $\pi : S_0, A_0, R_1, S_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    for step in episode do
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[G_t - \hat{v}(S_t, \boldsymbol{w})]\nabla\hat{v}(S_t, \boldsymbol{w})$
return $\hat{v} \approx v_*$

5.2.2 Semi-gradient Temporal-Difference and SARSA

By bootstrapping the targets, all depend on the current value of the weight vector $\boldsymbol{w}_t$, which implies that they will be biased and the method is not a true gradient descent. In fact, the passage in (5.2) relies on the target being independent from $\boldsymbol{w}_t$, and therefore is not valid if a bootstrapping estimate were used in place of $v_\pi(S_t)$. Therefore bootstrapping methods are not instances of true gradient descent [Barnard, 1993] and, since they take into account only a part of the gradient, are therefore called *semi-gradient methods*.

Despite the fact these methods don't converge as robustly as pure gradient descent ones, they do converge reliably in some cases, and offer important advantages that make them often preferable. One reason is that they generally have a significantly higher convergence rate, and they allow for continuous and online policy learning, without waiting for the end of an episode. A common semi-gradient method is the *semi-gradient TD(0)*, which uses $U_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, \boldsymbol{w})$ as its target. The pseudocode for this algorithm is shown in Algorithm 14.

Algorithm 14: Semi-gradient One-Step Temporal Difference

Inputs: the policy to be evaluated π, a differentiable function $\hat{v}$
Parameters: step size $\alpha > 0$
Initialization: weight vector $\boldsymbol{w}$ arbitrarily

for each episode do
  initialize S
  for step in episode do
    $A \leftarrow \pi(\cdot|S)$
    observe R, S'
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[R + \gamma\hat{v}(S', \boldsymbol{w}) - \hat{v}(S, \boldsymbol{w})]\nabla\hat{v}(S, \boldsymbol{w})$
    $S \leftarrow S'$
return $\hat{v} \approx v_*$

The extension of these methods to approximate the action-value function is immediate. In this case is the function $\hat{q} \approx q_\pi$ which is parametrized, as the state-value function before, with a weight vector $\boldsymbol{w}$, but now the update target U_t can be any approximation of $q_\pi(S_t, A_t)$. The general gradient descent update for the action-value function is:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha[U_t - \hat{q}(S_t, A_t, \boldsymbol{w})]\nabla\hat{q}(S_t, A_t, \boldsymbol{w}) \tag{5.4}$$

The function approximation version of the SARSA algorithm, for example, uses the following update rule:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha[R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \boldsymbol{w}_t) - \hat{q}(S_t, A_t, \boldsymbol{w}_t)]\nabla\hat{q}(S_t, A_t, \boldsymbol{w}_t) \tag{5.5}$$

which is called *Semi-gradient SARSA*. To form an efficient control method, there is need to couple action-value function approximation with techniques for policy improvement and action selection. If the action set is discrete and not too large, it is possible to compute $\hat{q}(S_t, a, \boldsymbol{w})$ for each available action a in the current state S_t and then find the greedy action $A_t^* = \arg\max_a \hat{q}(S_t, a, \boldsymbol{w}_{t-1})$. Policy improvement is then done by changing the estimation policy to a soft approximation of the greedy policy, such as the ε-greedy policy.actions are selected according to this same policy. The pseudocode for the corresponding algorithm is reported in Algorithm 15

Algorithm 15: Semi-gradient SARSA

Inputs: a differentiable function $\hat{q}$
Parameters: step size $\alpha > 0$
Initialization: weight vector $\boldsymbol{w}$ arbitrarily

for each episode do
  initialize S, A
  for step in episode do
    observe R, S'
    if S' **is terminal then**
      $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[R - \hat{q}(S, A, \boldsymbol{w})]\nabla\hat{q}(S, A, \boldsymbol{w})$
      break
    $A' \leftarrow \hat{q}(S', \cdot, \boldsymbol{w})$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[R + \gamma\hat{q}(S', A', \boldsymbol{w}) - \hat{v}(S, A, \boldsymbol{w})]\nabla\hat{q}(S, A, \boldsymbol{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$
return $\hat{v} \approx v_*$

5.3 Policy Gradient Methods

As it is possible to learn a state-value or an action-value function, is possible to learn a policy in the same way. The idea is the same as in function approximation, but this time it is parametrized the policy instead, with a policy parameters vector $\boldsymbol{\theta}$.

In order to learn the best policy, it is possible to use the gradient descent as described before, by taking into consideration the gradient of a performance measure $J(\boldsymbol{\theta})$ with respect to the policy parameters vector $\boldsymbol{\theta}$. Since this time we are actually trying to maximize the performance, the update rule will be ascending the function J:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\widehat{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)} \tag{5.6}$$

where $\widehat{\nabla J(\boldsymbol{\theta}_t)}$ is the stochastic estimate of the gradient of the performance measure. All methods that follows this general schema are called *policy gradient methods*.

In these methods, the policy can be parametrized in any way, as long as $\pi(a|s, \boldsymbol{\theta})$ is differentiable with respect to the parameters. If the action space is discrete and not too large, then an easy parameterization is to form some numerical preferences $h(s, a, \boldsymbol{\theta})$ for each state-action pair. The action with the highest preferences in each state are given the highest probabilities of being selected, for example by using the soft-max rule:

$$\pi(a|s, \boldsymbol{\theta}) = \frac{\exp(h(s, a, \boldsymbol{\theta}))}{\sum_b \exp(h(s, b, \boldsymbol{\theta}))} \tag{5.7}$$

The action preferences themselves can be parametrized arbitrarily. For example, they might be computed using a deep artificial neural network, such as the ones described in Chapter 2, where $\boldsymbol{\theta}$ is the vector of all the connection weights of the network.

5.3.1 Monte Carlo Policy Gradient (REINFORCE)

The stochastic gradient descent method requires a way to obtain samples, such the expectation of the sample gradient is proportional to the actual gradient of the performance measure as a function of the parameter. Furthermore, the sample gradients need only be proportional to the gradient because any constant of proportionality can be absorbed into the step size α, which is arbitrary. The *policy gradient theorem* gives an exact expression proportional to the gradient:

$$\begin{aligned} \nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla \pi(a|s, \boldsymbol{\theta}) \\ &= \mathbb{E}\left[\sum_a q_\pi(S_t, a) \nabla \pi(a|S_t, \boldsymbol{\theta}) \right] \end{aligned} \tag{5.8}$$

where μ is the on-policy distribution under π. It is then possible to define a basic stochastic gradient descent update as:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \sum_a \hat{q}(S_t, a, \boldsymbol{w}) \nabla_{\boldsymbol{\theta}} (a|S_t, \boldsymbol{\theta}) \tag{5.9}$$

which is often called an *all-actions* method because it involves all of the possible actions.

A better method is the *REINFORCE* algorithm, which is derived by introducing A_t in (5.8), by replacing a sum over the random variable's possible values by an expectation under π, and then sampling this expectation. This is done by introducing the weighting

term $\pi(a|S_t, \boldsymbol{\theta})$. With that, the gradient has now the form:

$$\begin{aligned}\nabla J(\boldsymbol{\theta}) &= \mathbb{E}_\pi \left[\sum_a \pi(a|S_t, \boldsymbol{\theta}) q_\pi(S_t, a) \frac{\nabla \pi(a|S_t, \boldsymbol{\theta})}{\pi(a|S_t, \boldsymbol{\theta})} \right] \\ &= \mathbb{E}_\pi \left[q_\pi(S_t, A_t) \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \right] \\ &= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(a|S_t, \boldsymbol{\theta})}{\pi(a|S_t, \boldsymbol{\theta})} \right]\end{aligned} \tag{5.10}$$

where G_t is the classic return. The final expression in the brackets is exactly what we were searching for. A value that can be sampled on each time step and whose expectation is equal to the gradient. This allows us to write the REINFORCE update rule:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)} \tag{5.11}$$

Here the increment is proportional to the product of a return G_t and the gradient of the probability of taking the action actually taken, divided by the probability of taking that action. This vector is the direction in parameter space that most increases the probability of repeating the action A_t on future visits to state S_t. The update increases the parameter vector in this direction proportional to the return, and inversely proportional to the action probability. The former makes sense because it causes the parameter to move most in the directions that favor actions that yield the highest return. The latter makes sense because otherwise actions that are selected frequently are at an advantage (the updates will be more often in their direction) and might win out even if they do not yield the highest return.

It should also be noted that the REINFORCE algorithm uses the complete return from time t, which includes all the rewards until the end of the episode. In this sense REINFORCE is a Monte Carlo method. Pseudocode of this algorithm is reported in 16. Here the fraction $\frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)}$ is written in the compact form $\nabla \ln \pi(A_t|S_t, \boldsymbol{\theta}_t)$, which is usually referred to as the *eligibility vector*.

As a stochastic gradient method, REINFORCE has good convergence properties. By construction, the expected update over an episode is in the same direction as the performance gradient. This assures an improvement in expected performance for sufficiently small α, and convergence to a local optimum under standard stochastic approximation conditions for decreasing α. However, as a Monte Carlo method REINFORCE may be of high variance and thus produce slow learning.

Algorithm 16: Monte Carlo Policy Gradient (REINFORCE)

Inputs: a differentiable policy parametrization $\pi(a|s, \boldsymbol{\theta})$
Parameters: step size $\alpha > 0$
Initialization: policy parameter $\boldsymbol{\theta}$ arbitrarily

for each episode do
  `episode` $\leftarrow$ new episode following $\pi : S_0, A_0, R_1, S_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  for step in episode do
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$
return $\hat{\pi} \approx \pi_*$

REINFORCE with Baseline

The policy gradient theorem (5.8) can be improved by including a comparison of the action-value function with an arbitrary baseline $b(s)$:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a (q_\pi(s, a) - b(s)) \nabla \pi(a|s, \boldsymbol{\theta}) \tag{5.12}$$

This baseline can be any function as long as it does not depend on a. In fact, this guarantess that the equation remains valid because, when computing the gradient, the value of the subtracted quantity is zero:

$$\sum_a b(s) \nabla \pi(a|s, \boldsymbol{\theta}) = b(s) \nabla \sum_a \pi(a|s, \boldsymbol{\theta}) = b(s) \nabla 1 = 0 \tag{5.13}$$

The corresponding update rule is a new version of the REINFORCE algorithm, which now includes a generic baseline:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha (G_t - b(S_t)) \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)} \tag{5.14}$$

Since the baseline could be uniformly zero, this is a strict generalization of REINFORCE. In general, by proper chosing the baseline, it is possible to leave the expected value of the update unchanged, but it can still have a large effect on its variance.

The most obvious choice for the baseline is to use an estimate of the state-value $\hat{v}(S_t, \boldsymbol{w})$ and, since REINFORCE is a Monte Carlo method for learning the policy parameter $\boldsymbol{\theta}$, it is immediate to also use a Monte Carlo method to learn the state-value weights $\boldsymbol{w}$. The pseudocode for this algorithm is given in Algorithm 17.

Algorithm 17: REINFORCE with Baseline

Inputs: a differentiable policy parametrization $\pi(a|s, \boldsymbol{\theta})$, a differentiable state-value function parametrization $\hat{v}(s, \boldsymbol{w})$

Parameters: step size $\alpha^{\boldsymbol{w}} > 0$, $\alpha^{\boldsymbol{\theta}} > 0$

Initialization: policy parameter $\boldsymbol{\theta}$ arbitrarily, weight vector $\boldsymbol{w}$ arbitrarily

for each episode do

  `episode` $\leftarrow$ new episode following $\pi : S_0, A_0, R_1, S_1, \ldots, S_{T-1}, A_{T-1}, R_T$

  for step in episode do

    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$

    $\delta \leftarrow G - \hat{v}(S_t, \boldsymbol{w})$

    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}} \delta \nabla \hat{v}(S_t, \boldsymbol{w})$

    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

return $\hat{\pi} \approx \pi_*$

5.3.2 Actor-Critic Methods

When the state-value function, or similarly the action-value function, is not used only as a baseline $b(s)$, but also to actively bootstrap the value, here we are faced with a new class of algorithms, the *Actor-Critic algorithms*.

This class algorithms are composed of two parts: the first one, which is called the "Critic" part, estimates the value of the state (or action) the agent is in (or has just taken), whilst the second one, the so-called "Actor" part, actually updates the policy distribution in the direction suggested by the Critic. Despite the REINFORCE with baseline method learns both the policy function and the state-value function, it is not considered to belong to the Actor-Critic algorihtms, since its state-value function is not used for estimating the current value for a certain state, but only as a baseline for the state whose value estimate is being updated.

REINFORCE with baseline is an unbiased method and will converge asymptotically to a local minimum but, since it is a Monte Carlo method it may learn slowly and it is unconvenient to implement for online or continuing problems. The *One-Step Actor-Critic* methods replace the full return of the REINFORCE algorithm with a one-step return (while using a learned state-value function as the baseline):

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha(G_{t:t+1} - \hat{v}(S_t, \boldsymbol{w}))\frac{\nabla\pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)} \\ &= \boldsymbol{\theta}_t + (R_{t+1} + \gamma\hat{v}(S_{t+1}, \boldsymbol{w}) - \hat{v}(S_t, \boldsymbol{w}))\frac{\nabla\pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)} \\ &= \boldsymbol{\theta}_t + \alpha\delta_t\frac{\nabla\pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)}\end{aligned} \tag{5.15}$$

The pseudocode for this algorithm is reported in Algorithm 18.

Algorithm 18: One-Step Actor-Critic

Inputs: a differentiable policy parametrization $\pi(a|s, \boldsymbol{\theta})$, a differentiable state-value function parametrization $\hat{v}(s, \boldsymbol{w})$

Parameters: step size $\alpha^{\boldsymbol{w}} > 0$, $\alpha^{\boldsymbol{\theta}} > 0$

Initialization: policy parameter $\boldsymbol{\theta}$ arbitrarily, weight vector $\boldsymbol{w}$ arbitrarily

for each episode do

- Initialize S
- $I \leftarrow 1$
- **while** S **is not terminal do**
  - $A \leftarrow \pi(\cdot|S, \boldsymbol{\theta})$
  - Take action A, observe S', R (if S' is terminal, $\hat{v}(S', \boldsymbol{w}) = 0$)
  - $\delta \leftarrow R + \gamma\hat{v}(S_{t+1}, \boldsymbol{w}) - \hat{v}(S_t, \boldsymbol{w})$
  - $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}}\delta\nabla\hat{v}(S_t, \boldsymbol{w})$
  - $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I\delta\nabla \ln\pi(A_t|S_t, \boldsymbol{\theta})$
  - $I \leftarrow \gamma I$
  - $S \leftarrow S'$

return $\hat{\pi} \approx \pi_*$

5.3.3 Deep Learning Methods

In this chapter we have presented several function approximation algorithms for reinforcement learning, but how can these be "deep"? As briefly mentioned above, both the prediction and control methods described here are based on a parameterization of a function, in particular $\hat{v}(s, \boldsymbol{w})$ with respect to the weight vector $\boldsymbol{w}$ and $\pi(a|s, \boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$. Now, if we think of these approximation functions as neural networks, the $\boldsymbol{w}$ and $\boldsymbol{\theta}$ vectors become exactly the weights vectors of a single perceptron, as described in Chapter 2 (and then the weigths matrices in a multy-layer perceptron).

To better understand, let's take a practical example about the structure of a deep reinforcement learning setup. In particular, let's take exactly the last category of algorithms studied, the actor-critic ones, and see how they can be quickly implemented on a neural network. The Figure 5.1 shows an artificial neural network in a two-headed configuration. The first one, the one with more output units, is the one that will represent the function approximation of the policy $\hat{\pi}$, while the second one, constituted by only one unit, will represent the function approximation for the value function $\hat{v}$. Since this is a multi-layered model, obviously the parameter vectors $\boldsymbol{w}$ and $\boldsymbol{\theta}$ have now become matrices. This, however, this does not change the way the model works, which will use the same algorithm described in Algorithm 18 to train them.

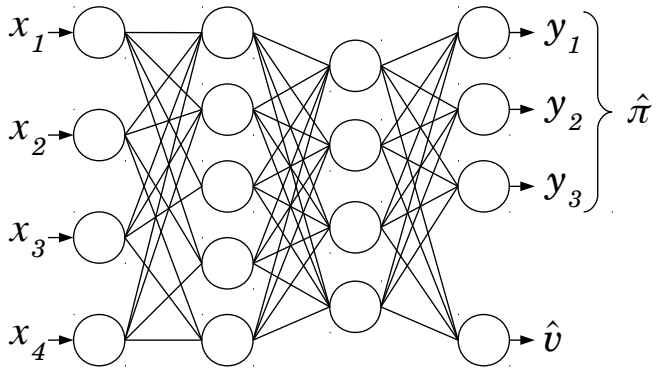


Figure 5.1: Deep neural network in a reinforcement learning setup.

Finally, there is one last interesting thing to discuss about: it may seem strange in fact to consider a neural network with two separate output layers. Typically we have a policy network that provides a probability distribution over actions and another that gives a value estimate of the state. If we have n actions, then the policy network yields

n values that sum to 1, while the value network always yields a single value, but, with a two-headed network we wind up getting a shared body of layers that branch off into these separate heads. In order to update this, we have to pass the gradients coming out of both heads, so the body gets updated according to the gradients of both the policy network and the value network. What these networks are actually trying to do, is in fact just to learn to approximate some functions based on the data we provide. So all we're doing is adjusting the parameters of the network to best approximate this function, regardless of what the output is.

5.4 State of the Art of Reinforcement Learning

Most of the algorithms described in the previous sections are related to mathematical optimization theories developed during the late 1950s. However, it has only been in recent years that reinforcement learning has undergone a considerable rediscovery and diffusion, mainly due to the improvement of the hardware available used for computation.

5.4.1 Multi-agent Reinforcement Learning

A novelty introduced thanks to parallel computing development, is the possibility of using a multi-agent system to train a common network. In a multi-agent framework, many learrning agents are equipped with a local copy of the global network, and interact with the same environment (or with each other, depending on the task). After a training period, each worker then updates the global network based on what has been learned. Many algorithms already presented can be developed asynchronously [Mnih et al., 2016], such as the policy gradient methods explained in Section 5.3. An example of the global architecture of the algorithm is represented in Figure 5.2.

It was shown that asynchronous algorithms are capable of training a neural network in a stable manner as well as synchronous ones, and even better in some occasions. What has been found is that using parallel actor-learners to update a shared model has a stabilizing effect on the learning process. The asynchronous models also train faster on a multi-core CPU with respect to the synchronous ones trained on a GPU, this resulting in better performances in the same training time.

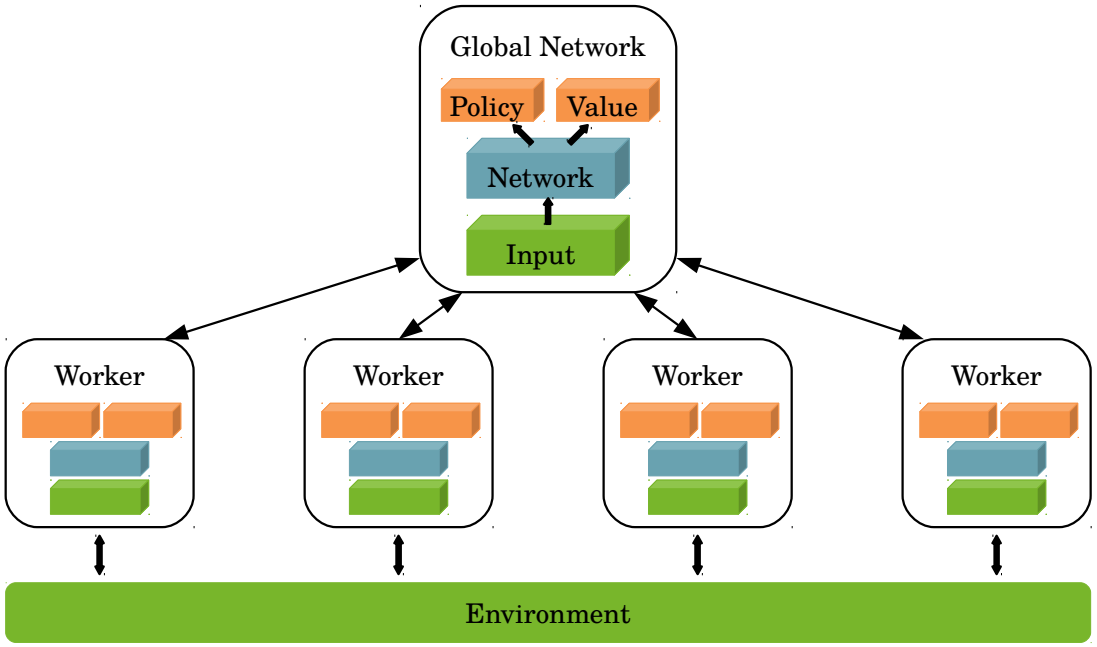


Figure 5.2: Asyncronous Actor-Critic framework.

5.4.2 Learning to Play Games

The area where reinforcement learning has been most successful is undoubtedly that of games and videogames, where agents are trained to play at beyond the human level. In [Mnih et al., 2013] and [Mnih et al., 2015], for example, a convolutional neural network was developed , such as the ones described in Chapter 2, which was trained with a variant of a reinforcement learning algorithm called *Q-learning* (not reported here). This network has proven to be capable of learning how to play seven different Atari 2600 games, outperforming previous approaches and surpassing human experts in most of them, using only the raw pixels of the screen as input.

More recently, Google Deepmind's AlphaGo reached the great milestone of defeating a Go champion on a standard-sized table [Silver et al., 2016]. The game of Go has long been viewed as the most challenging of classic games for artificial intelligence, owing to its enormous search space and the difficulty of evaluating all the board positions and possible moves. In the AlphaGo proposed model, two deep convolutional neural networks are used: the first evaluates the board position, whilst the other one selects moves (as in

the Actor-Critic approach of the previous chapter). This two networks were trained using a combination of supervised learning (from human-expert games) and reinforcement learning (from self-played games). In the next improved model AlphaGoZero [Silver et al., 2017] instead, the two networks were merged into a single network, and the algorithm was based solely on the reinforcement learning approach, without human data. This last model was proven to achieve even better results than the previous one, winning a series of mathces $100 - 0$ against the first model.

Finally, only during the last year significant results were also achieved in the field of real-time-strategy (RTS) video games, in particular the game StarCraft II, played by Google Deepmind's AlphaStar [Vinyals et al., 2019]. Although the significant successes in the field, until now artificial intellicenge techniques have struggled to cope with the complexity of RTS games. AlphaStar behaviour is generated by a deep neural network (which is actually a combination of different architectures, from feedforward to recursive) that receives input data from the raw game interface, and outputs a sequence of instructions that constitute an action within the game. The neural network was initially trained by supervised learning, which allowed AlphaStar to learn by imitation, and then it learned by playing against itself in a *multi-agent* reinforcement learning process. In the end, AlphaStar was able to beat a top professional player in a series of matches $5 - 0$.

Part III

Industrial Applications

Chapter 6

Methods and Models

Before explaining in detail the problem faced during my thesis work, and the methods used to solve it, we are going to start with an easier toy problem: the arcade game Snake. This, despite at first glance may seem to have little to do with the main problem, it will actually serve as a starting point for the study and development of the policy gradient algorithm used afterwards.

We move on then to describe the problem of recognizing defects on fruits, and briefly discuss what the current approach is, and how to overcome the difficulties that emerge from this. Then the first attempts to model the problem are presented, and we'll go and see where they proved to be adequate or where they could be improved. In the end, the final problem framework and methods are presented, with particular attention to the dataset generation, used to create samples and assess the applicability of the model.

6.1 Starting with a Toy-Problem: The Snake Game

To correctly implement the some deep reinforcement learning algorithm within the code, we first started by testing it on a toy-problem: the arcade game Snake. This is a game in which the player controls a line of squares (the snake) on a bordered plane, and the goal is to collect as many squares of another color (the food) as possible, increasing the length of the snake as they get eaten. The game is lost when the player eats himself or ends up off the playing field. A graphical representation of the game is reported in Figure 6.1, which shows the snake in three different phases of the game with a different length. The whole Snake game environment was developed using Python.

Figure 6.1: Different time steps of the Snake game.

Actually, the algorithm implemented within the Snake environment is the Actor-Critic algorithm, whilst the one implemented in the main problem faced will be the REINFORCE one. This is due to the fact that, while the former turns out to be a non-episodic task (as the snake could never lose during the game), the latter has well-defined episodes (the analysis of a single fruit), and it is therefore useless to use the value function also to bootstrap the state-values for each update. However, this does not cause major differences in the implementation of the algorithm since they are quite similar, and the transition from one to another was immediate.

6.1.1 Network Architecture and Input Mapping

In order to beat the game, a feedforward neural network was implemented the TensorFlow package. This should have been deep enough to give rise to sufficiently complex behavior, and therefore a network with hidden layers of shape $40 - 32 - 24 - 12$ was created, each of those with $\tanh(x)$ as the activation function.

The network accepted an 8-dimensional input vector, which coded the essential information in order to correctly play:

- `input[1-3]`: the distance between the snake head and the boundaries, with respect to the direction of the movement (left, front, right), normalized in the interval $[0, 1]$;
- `input[4-6]`: the distance between the snake head and the snake body (in present along the direction), with respect to the direction of the movement (left, front, right), normalized in the interval $[0, 1]$;

- `input[7]`: the Manhattan distance between the snake head and the food square, normalized in the interval $[0, 1]$;
- `input[8]`: the angle between the snake head and the food square, with respect to the direction of the movement, rescaled from the interval $[-\pi, \pi]$ to $[-1, 1]$.

Random noise was then added to each of the distances, in order to speed up the convergence of the algorithm, in the range $[-1/s, +1/s]$ (where s was the size of the board).

At the other side, the network output a 3-dimensional vector, which represented the probability of turning left, continuing straight or turning right. This vector was then normalized through the softmax(x) function, and then a random action was taken, following the obtained distribution.

To train the network for this problem, as well as for the main one, we used the Adam optimizer, explained in Chapter 1, with a learning rate of 10^{-4}, to minimize the loss function explained later in this section.

6.1.2 Rewards and Loss Function

In order to work the Actor-Critic algorithm needed to estimate the value of a certain state and, in order to so, there was need to define the rewards and the discount factor. As a reward, the agent earned a reward of $+1$ for each fruit he has been able to eat, and a negative reward of -1 if it had eaten itself or had exited the game boundaries.

To prevent the game from running indefinitely, the maximum number of steps that the snake could have taken without eating anything was set to four times the size of the board. The discount factor γ was set equal to 0.9, since it was not really important when a snake ate some food, as long as it remained alive. For lower value in fact, in certain occasions the agent might prefer moving straight to the food square without taking into consideration the risk of hitting a part of itself. However, it was not fixed higher in order to prevent diverging values in the state-value function approximation.

In order to apply the gradient descent rule in TensorFlow, there is need of the analogous loss function for the update rule of Algorithm 18, defined as:

$$\mathcal{L}_\pi = -\log\left(\pi(A_t|S_t, \boldsymbol{\theta}_\pi)\right)\delta_t - \beta \cdot H\left(\pi(A_t|S_t, \boldsymbol{\theta}_\pi)\right) + \alpha \cdot \left(\hat{v}(S_t) - G_t\right)^2 \tag{6.1}$$

Let's take a moment to comment this formula. The first part is the standard policy gradient loss function: the δ_t is the TD-error at step t, and the policy gradient loss

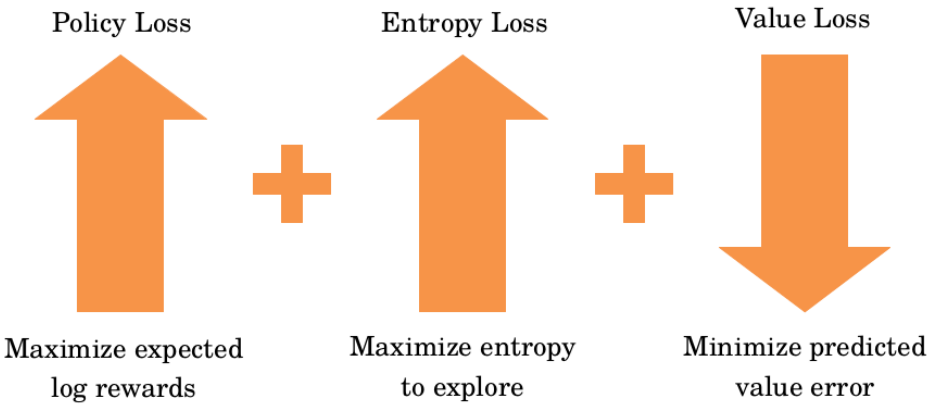
function is the error times the log of the probability of the action a taken at t.
To this, we also subtract the entropy term, which is scaled by a factor β (relatively small, usually $\sim 10^{-2}$) because this ought not dominate the loss we get from the rest of our loss function. The idea here is to use entropy to encourage further exploration of the model. Recall that the output of a network with policy gradient is a probability distribution over possible actions. If we have three actions to choose from, the softmax layer will give us three probabilities and, in order to prevent premature convergence, the we want to penalize the algorithm for being overconfident in a certain action by increasing the loss through the entropy term. In essence we're trying to adjust our probabilities to gain more information. It's important to note that we're taking the negative of our loss function here, thus subtracting our entropy term. This might be confusing at first, so let's think carefully about what our loss function is designed to do.

In short, our loss function is designed to increase the probability that we'll take high reward actions in the future. We want to change the gradient of our network to push that probability in the right direction. The first term for the policy gradient loss is the log probability times our TD-error, which can be thought of as the expected amount we outperform (or under perform if δ is negative) our baseline estimate of that state. We want to maximize this, however because deep learning frameworks like TensorFlow don't have the ability to maximize a function, we then minimize the negative of our loss, which is mathematically equivalent. We desire this first term then to be as negative as possible. Because the entropy is a positive value, we subtract it from this loss. This means that the loss is more negative when the output is high in entropy relative to the low entropy values. In this sense then, we're telling our network to update itself to favor high entropy values over low entropy to encourage exploration.

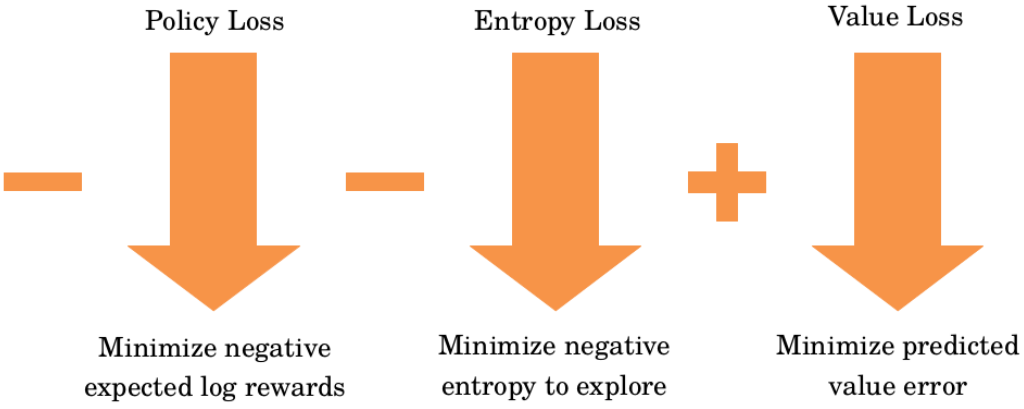
Because we're shoving all of these different gradients through the same network, we need to include the value loss as well. We'll just take the mean squared error (MSE) between the value prediction and the discounted rewards. To help learning along, we'll also introduce a scaling function for the value function gradients called α. This will be used to get the gradients to roughly the same order of magnitude so that one of our gradients doesn't dominate the others.

These three terms all work together to influence the behavior of the algorithm. At the end of the day, this algorithm isn't explicitly trying to maximize its reward, it's trying to minimize or maximize its loss function. Looking at it closely, these three values are working together for different purposes as shown in Figure 6.2(a).

What we see in the image is that we want to increase our policy loss and entropy loss while decreasing the value loss because the first corresponds to our expected rewards, which we obviously want to maximize in most cases. The entropy loss here is like as a "bonus" value that the algorithm receives for exploring, because the entropy will always be some non-zero value and increase the loss, but it adds more to the total loss by having a higher entropy. Then the value ought to be minimized because it is decreasing the predicted value estimation error for each state. As stated above, because it is only possible to minimize functions in TensorFlow, we adjust the signs so that we're minimizing all of our values during the training (see Figure 6.2(b)).



(a) Original optimization goal.



(b) Actual optimization goal.

Figure 6.2: Loss function optimization goal.

6.1.3 Results

As said before, the objective of the implementation of this project was simply to study the functioning of the policy gradient algorithms and how to apply them to concrete problems. For this reason, we are not interested in the performance of the algorithm nor in finding a way to improve it, but only that it has been developed correctly.

In order to see this, we trained the model over $250k$ games, each of size 10×10 (see Figure 6.1). Thanks to the Actor-Critic algorithm, which updates the network every 5 steps or when the game is over, the agent quicly learns how to survive and then how to grow in length without dying. The trend of the rewards is then reported in Figure 6.3.

Figure 6.3: Snake training rewards.

Since the total loss function is the composition of several losses from (6.1), in general the graphic representation is not immediately understandable as for the reward. The trends, however, are still reported in Figure 6.4 for the sake of completeness.

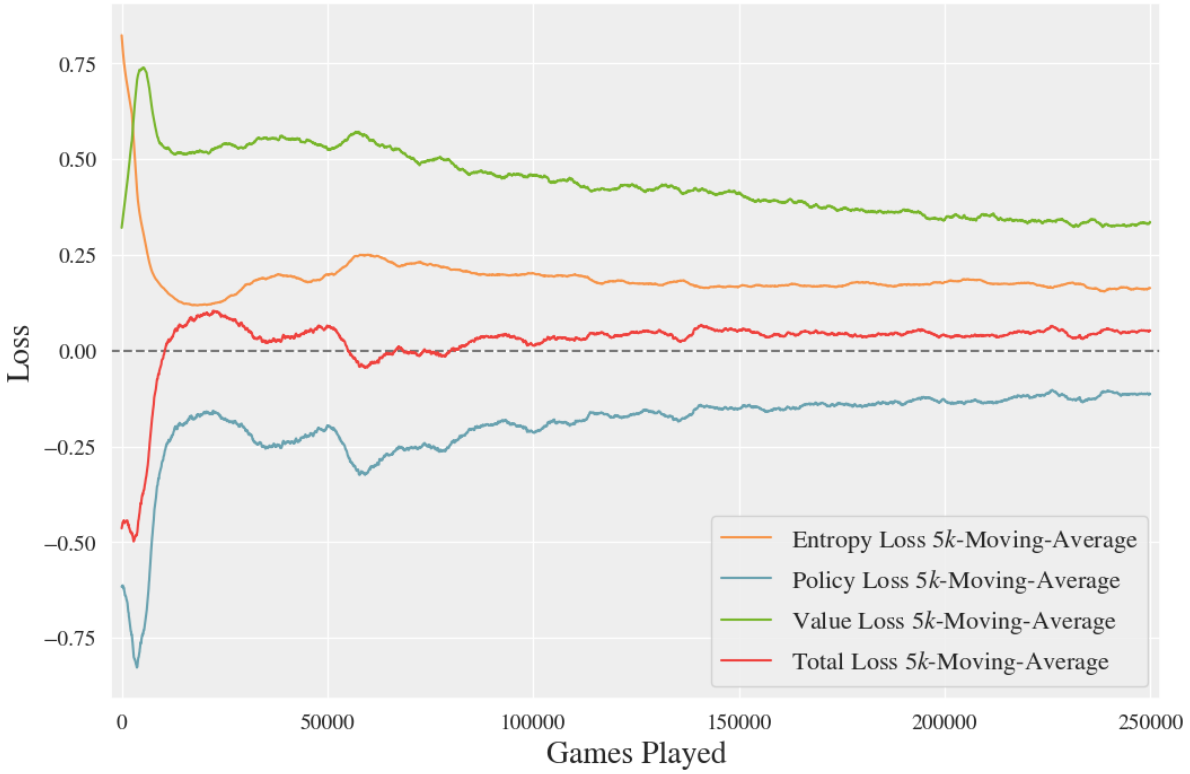


Figure 6.4: Snake training losses.

Here it is possible to see that the entropy loss starts from a rather high value, and gradually decreases, a sign that the network is more and more secure of the choices it makes, and therefore discourages exploration later in the training process. The value loss instead starts from a lower point (since in the start all the values were fixed randomly around ~ 0.1) and then increases because the network started correct the values. Once reached its maximum, then it decreases again as the model starts to make more correct estimations. Finally, the policy loss is following a more or less specular path of the value loss, but is instead increasing. In this way the expected log rewards are maximized, as explained before in 6.1.2.

At the end, the agent was tested on $25k$ other games, without improving the network, in order to estimate the expected reward for a single game as:

$$26 \pm 8$$

which leads us to believe that the algorithm was correctly implemented.

From Games to Industrial Applications

Seeing how reinforcement learning techniques can be successfully applied to the gaming world, mostly a simplification of the real world, one immediately wonders if these cannot also be applied to real life situations with the necessary simplifications, and in particular referring to industrial applications [Hammond, 2017].

The premises are very promising, since the goal of reinforcement learning framework is simply to learn how to map observations and measurements to a set of actions while trying to maximize some long-term reward and, in simple terms, it reduces in finding the optimal sequences of decisions. Therefore, at least in theory, it is possible to model any aspect of real life, not just games, as a reinforcement learning problem.

Even the fields of application could then be the most varied. Let's only think about problems of text and speech comprehension, image and video analysis, network optimization, process planning, demand forecasting, robotics and automation, fleet logistics, froduct design, service availability and many others.

6.2 Industrial Scenario

The problem faced during my thesis work mainly concerns the agri-food industry, with particular reference to the sector that produces and selects the fruits to be sold.

All of my work has been co-supervised by Dr. Matteo Roffilli, CEO at Bioretics s.r.l., a small cutting-edge company focused on research and development in artificial vision, who has proposed me to find a way to improve their approach to fruit selection, in order to better select the defects detected on different kind of fruits.

6.2.1 Optical Sorting

In the food industry it is very common to use machinery called *optical sorters*, which allows to recognize color, shape, structural properties and chemical composition of some solid products, using a combination of cameras and lasers. These sorters are generally used to compare the objects in question with some pre-established filtering rules, in order to accept or reject them from the production chain. Unlike manual sorting, which is subjective and often inconsistent, optical sorting provides an objective evaluation method, helping to improve the quality of the product, maximize throughput and increase yields while reducing labor costs.

In general, optical sorters are composed of four steps: the feed system, the optical system, an image processing software and the separating system. The main purpose of the feed system is to distribute evenly the product, in order to present it to the optical system without clumps and at a constant velocity. Then, the optical system uses lights and sensors to create some images and data to be passed on to the image processing system. This compares objects with some previously defined accept/reject thresholds to classify the objects and actuate the separation system. The separation system then selects and mechanically picks up the defective products from the suitable ones.

Sensors and Software

In order to correcly classify and separate objects, optical sorters use a combination of lights and sensors to illuminate and capture images, which will be processed and are used to make accept/reject decisions. There are multiple possible combination of cameras and lasers sensors which can be made. These can in fact be designed to function within the visible light wavelengths, as well as the infrared (IR) and the ultraviolet (UV) spectrum. The optimal wavelengths for each application maximize the contrast between the objects to be separated. Cameras and laser sensors can differ in spatial resolution, with higher resolutions enabling the sorter to detect and remove smaller defects.

Simple monochromatic cameras are able to detect several shades of gray and can be very effective when sorting products with high contrast defects. If the defects present subtle shades instead, it might be better to use more sophisticated color cameras, which are capable of detect millions of colors.

While cameras capture information based primarily upon object's reflectance, lasers are able to extract structural properties in addition to determining differences in color. This property makes lasers ideal for detecting a wide range of foreign material, both organic and inorganic, even if they are the same color as the product.

Once the sensors have captured several images, the image processing software starts to manipulate the data and extracts and categorize some specific features. The effectiveness of this image processing step lies in the development of algorithms that maximize the accuracy of the sorter, while being fast and easy to execute.

Object-based recognition is a classic example of software-driven intelligence. It allows the used to define a defective product based on where a defect lies on the product and/or the total defective surface area of an object.

6.2.2 Current Approach

The faced problem consisted in selecting whether or not a fruit was suitable for sale, based on the presence or absence of different classes of defects on it. In order to do so, a fruit candidate is passed on a conveyor belt inside an optical sorter. Here four cameras (two visible light cameras and two infrared cameras) aquire several shots of the fruit at different time steps. While the visible light cameras just aquire a normal shot of the fruit, the IR cameras are used to create a virtual mask of the fruit, which allows to separate it from the background for a better analysis.

The obtained image is then fed into a convolutional neural network, which is pre-trained to recognize the different parts of the fruit and the present defects. A specially written software also extrapolates some useful information about the fruit, such as the length and the width, the major/minor axis ration, and so on. Each part recognized by the software is then labeled, such as the stalk or the color of the fruit, and each defect is associated with a class number, describing its type, and a probability of correcteness of the identification. An example of the overall detection process is reported in Figure 6.5.

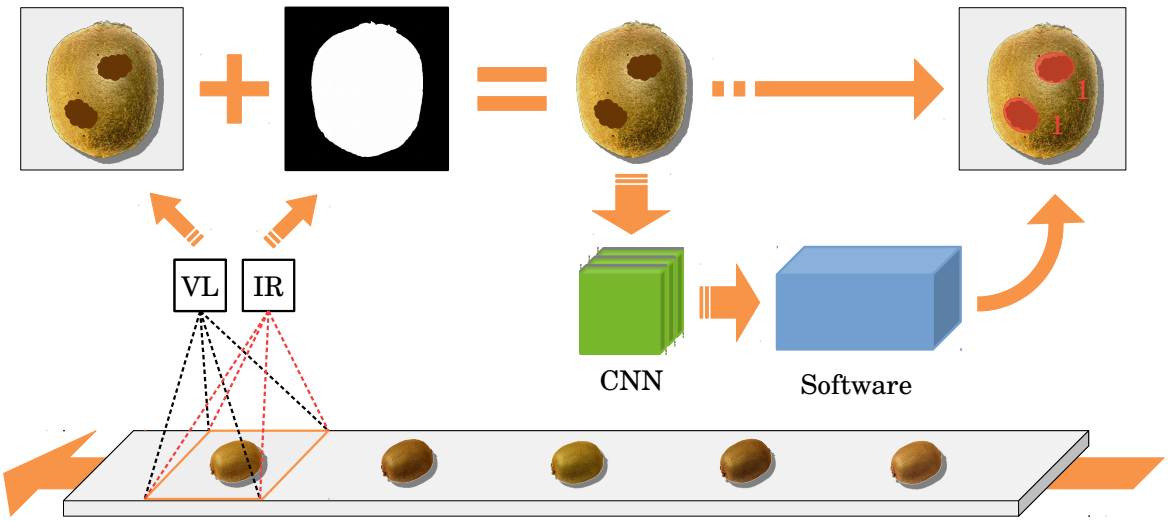


Figure 6.5: Optical sorter detection process.

The software then uses the information obtained to choose whether to discard the fruit or keep it in terms of defects present, but based only on the result of the worst shot. The software in fact does not take into consideration the rotation of the fruit, and does not attempt to match the defects between the different shots of the same fruit. Another example, showing three shots of the same fruit acquired at different time steps, is reported in Figure 6.6, showing also the information extracted by the CNN.

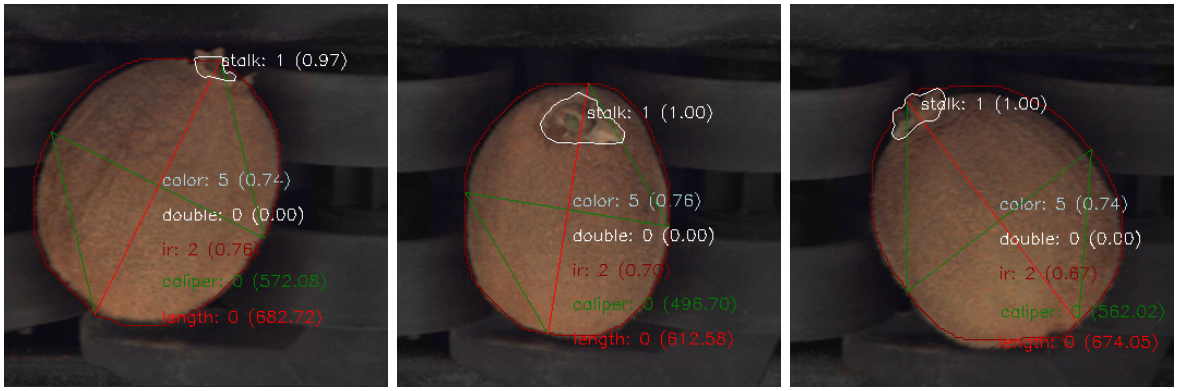


Figure 6.6: Three shots of the same fruit with CNN information.

Using Reinforcement Learning to Track Defects

The approach proposed in this thesis is then aimed at taking into consideration the physical rolling process of the fruit, and in trying to link the same defects between different shots. In order to do this, it was decided to train a deep neural network, without going to bother particular complex structures (such as convolutional or recurrent ones), to compare the properties of the defects and to recognize when they are sufficiently similar to be identified as the same defect.

The use of a labeled output and supervised learning was the initially favored choice, but this would not have deviated from a traditional approach of developing a classification model for the defects, and which more importantly would not have taken the rolling process into account at all. The use of reinforcement learning instead, would have introduced the concept of system state (in this case the rotation of the fruit) which would probably have influenced the defect identification process. The proposal is therefore to use a simple feedforward neural network, but using an algorithm that takes into account the rolling process during the training phase.

Therefore, it is necessary for the network to consider the step of the rolling process in which the fruit is, thus it is needed that this is coded and supplied to the network by means of the input. The final input vector will therefore contain a combination of features that describe the difference between two defects, in order to compare them, and also features that describe the rolling state of the fruit.

The algorithm that would be optimal for this case, is an algorithm that takes into account the state of the system when it's time to take an action, and that uses it within the training process. The most immediate idea that comes to mind, is then to use one of the policy gradient algorithms explained at the end of Chapter 5. These in fact use a baseline to train the policy, such as the REINFORCE with Baseline (17) or the Actor-Critic (18) algorithms.

6.3 Modeling Defects and Fruits

Subsequent efforts, in order to solve the problem of fruit sorting, focused mainly on understanding how to model the rolling process and how to process the data obtained for an eventual identification phase. It should be noted in fact that this thesis work is inserted after the CNN elaboration phase reported in Figure 6.5, where defects have already been identified and their properties extracted.

Once it has been decided how the process of identifying the various defects should take place, various ways have been tried to simulate the shot acquisition process, as well as the simulation of the various properties of the defects on the fruit. Once the first results were obtained, the process was then improved, trying to create a dataset that would allow faster processing and greater customization capabilities.

6.3.1 Defects Comparison and Identification Process

When loaded, each defect is also accompanied with a set of useful information, obtained from the CNN and software configuration from Figure 6.5, such as:

- the *position* (x, y) of the defect normalized with respect to the size of the shot;
- the *circularity*, defined as:

$$c = \frac{4\pi \cdot \text{Area}}{\text{Perimeter}^2} \tag{6.2}$$

- the *eccentricity*, defined as the ratio of the focal distance over the major axis length;
- the *solidity*, defined as the ratio between the area of the region and the area of the convex hull surrounding it;

which will be used to compare the defects between each other.

It is now important to describe the process of comparing defects and the following process of identification separately, because while the former is actually used to train the model used, the latter is only proposed as an extra step for defects management.

Defects Comparison Process

Once created, these defects will be stored inside a fruit object, which will take care of applying the correct spatial transformations (simulating the rolling process) and presenting them in the correct temporal order of analysis (the shot number). Each defect will be in fact compared by the neural network with all the already analyzed defects from all the previous shots. For each comparison, the output of the neural network can be either *identical* or *different*. The agent then checks that the answer given by the network is correct with the ground truth and, in case of correct answer, it receives a reward of +1, otherwise -1. At the end of the analysis of each single fruit, the accuracy achieved by the model for the current fruit is calculated by simply rescaling the final average reward (bounded to be in the interval $[-1, 1]$ by construction) in the interval $[0, 1]$.

Identification Process

As far as the process of identifying the various defects is concerned, the matter becomes slightly more complicated. As the smallest element to be analyzed is the single defect, we want the program to be able to univocally identify these elements after each analysis process. To do so, each defect object is defined to have a universally unique identifier (UUID), which is represented by a string of 32 hexadecimal digits displayed in groups of the form $8 - 4 - 4 - 4 - 12$. This guarantees that there are more than $3 \cdot 10^{38}$ different possible UUIDs and therefore, when one is randomly assigned to a new defect, the probabilities to have the same UUID while not being actually the same are almost zero.

For each defects comparison, if the output is the label *identical*, the compared defect's UUID is stored in a list of possible UUIDs of the current defect. After the current defect has been compared with all the previous defects, the most frequently attributed UUID is

assigned to the defect. If the output label is always *different*, no UUID is ever attributed and a new one is then generated, because it means that the defect has no similar in the other shots. It is important to specify that a reward of ± 1 is earned for each label attribution and, only at the end, when all the defects have an assigned UUID (which may be correct or not) the network is updated with all the results. This is the main difference between the fruit framework and the Snake game, where the updates were made every few steps, regardless of whether the game was over or not. This overall process of UUIDs assigning is reported and described in Figure 6.7.

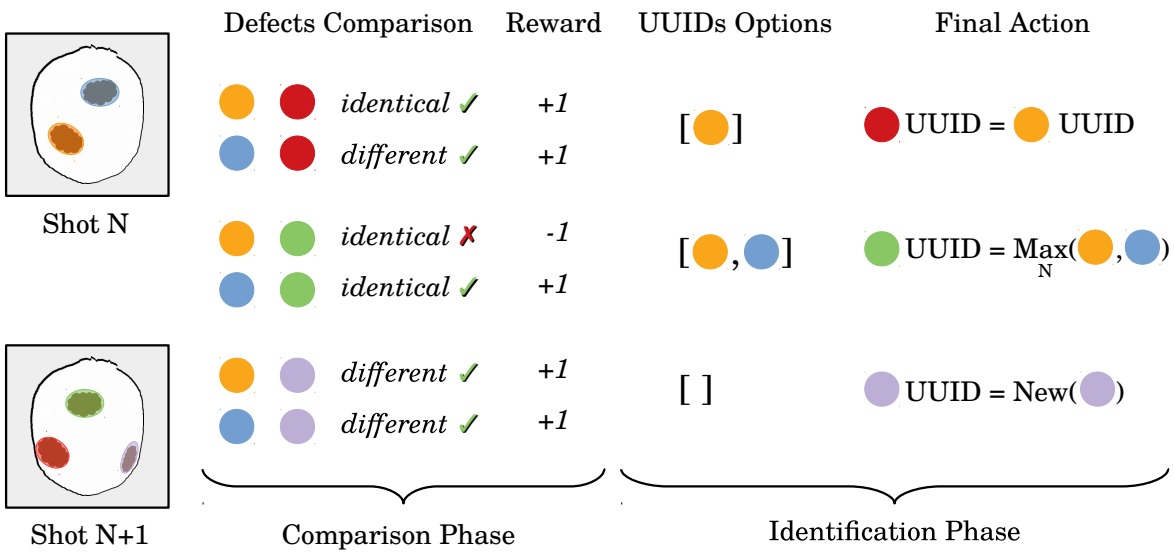


Figure 6.7: Defect comparison and UUIDs assignement processes.

Here the colours are used in order to better describe the comparison between defects of a shot with the previous ones. In this example, only the comparison with the immediately preceding shot is shown for simplicity but, actually, all the previous shots are taken into consideration. Also, in case of a tie between the frequencies of the UUIDs assigned, the resulting UUID will be a randomly chosen between the most frequent ones.

6.3.2 Rolling Process Simulation for Synthetic Dataset

Unfortunately, a dataset with the characteristics needed for the process described was not readily available, since no one contained all the information about the identification of the defects on a fruit. Therefore, it was necessary to artificially create it, but simulating the data as realistically as possible, to not affect the applicability in real situations.

Generating Shots

The initial intention was to fully simulate the process of acquiring shots and processing the properties of defects. For this purpose we wrote some code that allow the creation of points on a spherical surface, with many possibilities of scale and color customization. The surface was then rotated, and an image saved to simulate the shot acquisition. As an example, three generated shots at different time steps of the same fruit are reported in Figure 6.8, where it's immediate to notice the simulated rolling process.

Figure 6.8: Five simulated shots of the same fruit during the rolling process.

On the same fruit each defect is uniquely numbered and its position is always tracked, so it's possible to create a list with the correct id of the defects contained in the same shot, which will come in handy later in the training phase. After this, all the shots images are converted in greyscale and the defects properties are extracted using simple image segmentation algorithms.

This approach has proved to be very effective for the overall process simulation, but with several limitations. First of all, the size and shape of the fruits were not realistic because they were perfectly spherical and not customizable. Secondly, the generation of a very high number of images entailed very high costs in terms of memory and computation. Finally, the properties extraction process relied on very noise-sensitive labeling algorithms, with the risk of combining together some defects that were too close.

Generating Geometrical Data

The most important improvement made to the code was the transition from image creation to analytical simulation. Although it was initially difficult to write the part of the code that correctly computed the properties of the defects, this allowed for greater customization of the properties of the fruit and greatly speeded up both the dataset generation process and the training phase. The key idea here is to think of a single fruit as a relatively simple geometric entity, like an ellipsoid. Consequently, the rolling process can be modelled as a simple three-dimensional rotation of the fruit surface. Thinking then of the defects as simple points on this surface, the mathematical operations to be carried out are reduced to trivial matrices multiplications.

The general equation for an ellipsoid (centered in the origin) is:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1 \tag{6.3}$$

where a, b, c are the lengths of its principal axes. This equation also has a matrix form representation as a quadric:

$$\boldsymbol{x}^\top A \boldsymbol{x} = 1 \tag{6.4}$$

where A is a positive definite matrix, whose eigenvectors define the principal axes and the eigenvalues are the reciprocals of the squares of the semi-axes a^{-2}, b^{-2}, c^{-2}. This form allows us to easily compute a rotation around an axis and of an angle of our choice, by simply multiplying the matrix A with the corresponding rotation matrix R.

Therefore, to simulate a custom fruit, we just have to choose the lengths of the axes a, b, c and create its corresponding matrix. The coordinates of the defects on it, on the other hand, are only a list of points whose position will be tracked during the rotations.

Applying Defects

The next task is to generate the defects on the fruit, each with a different shape, size, and properties, and to correctly compute these during the images acquisition.

This is actually performed by projecting several defects onto the three-dimensional ellipsoid described before, that will be centered in the origin, and have axes a, b, c set as the length and caliper generated from the dataset. In particular two of them (those describing the caliper) will be the, in order to simulate a symmetrical fruit. The surface will then perform two rotation, the firts along the z-axis (perpendicular to the acquisition

plane) and the second one along its major axis. After this, all the points that have the coordinate $z > 0$ will be considered as acquired, and the properties of the fruit defects are evaluated. The difficult part of all of this, is to correctly compute each of the defect properties, such as area and perimeter, with taking into consideration the "cutting" effect of the shot acquisition. In order to do so, we're going to decompose each defect into a mesh grid, by performing a ncloud point Delaunay triangulation. This will guarantee that, even if some points will be missing, the properties will be computed based oly on the current points. A visualization of the process is reported in Figure 6.9.

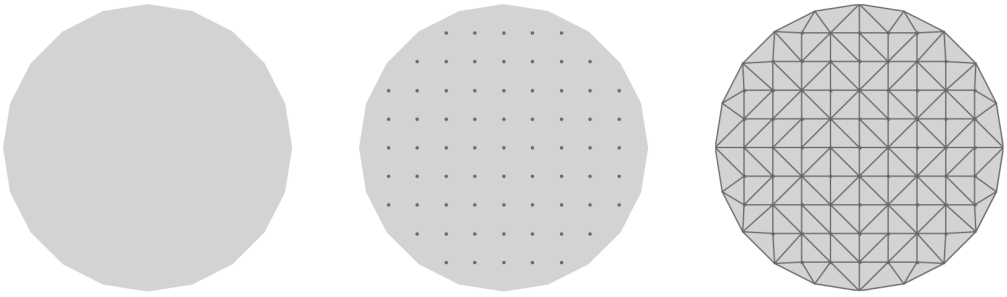


Figure 6.9: Defect mesh grid creation process.

Once a new defect of a certain shape is created, it is rescaled up to a random factor and projected onto the ellipsoid using the appropriate transformation in spherical coordinates. Obviously the defect properties won't be preserved during this projection, but the most important thing is that they will be preserved during the forward rolling process simulation. Some randomly generated kiwis are reported in Figure 6.10, each with four defects projected on the surface. Here the ellipsoid is represented as slightly transparent in order to see also the defects on the other side of the surface. It is also possible to see its three axis, where the green one is the major axis of rotation.

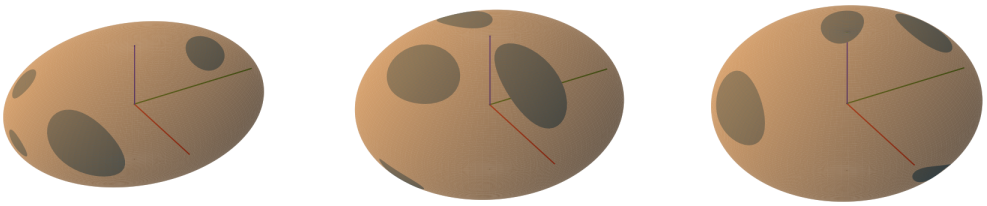


Figure 6.10: Three randomly generated fruits with four defects each.

6.4 Creating the Dataset

As said before, a dataset with exactly those characteristics was not immediately available. However, it was possible to retrieve data on the size and rolling process of real fruits, obtained through a process similar to the one reported in Figure 6.5. This has allowed us to create a custom dataset that, even if generated artificially, was based on data obtained from a real process, and therefore of immediate applicability.

6.4.1 The Real Dataset

The dataset containing real data was provided by Ser.mac s.r.l., a company collaborating with Bioretics, with measurements of 1713 kiwis. For each of these, 5 shots were acquired from two cameras, positioned one opposite to the other in a left-right configuration, for a total of 17130 entries. The measurements reported in the dataset indicated an ID of the fruit, the camera used, the view number, the positions of points used to calculate the length and the caliper, and of the point identified as the stalk (if recognized). An example of the images acquired for each fruit are reported in Figure 6.11 while an excerpt of the provided dataset is reported in Table 6.1. All the measurements are reported in pixels, with a known conversion factor of 3 px/mm.

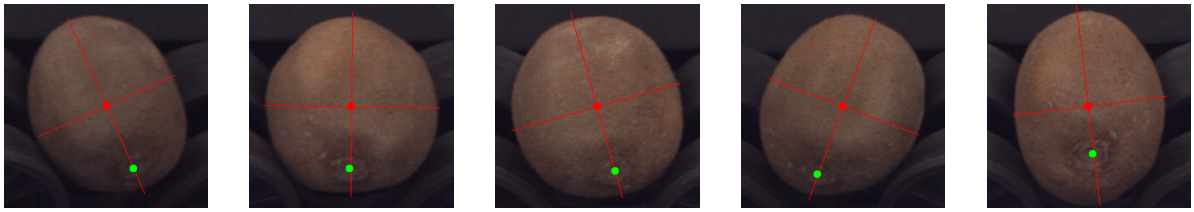


Figure 6.11: Five consecutive shots of the same fruit during the rolling process.

The raw data were then cleaned and parsed, in order to extract more useful informations. For simplicity of data analysis, the measurements obtained by the two cameras were treated as independent acquisitions and therefore, after the processes, the dataset was composed of 5 shots of 3418 kiwis, for a total of 17090 entries. An excerpt of this dataset is reported in Table 6.2. Here we can observe the calculated caliper and length of the fruit, alongside with the angle α, the rotation angle around the axis perpendicular to the conveyor belt (z-axis). The angle $\Delta\alpha$ is simply the difference between two consecutive angles, which therefore measures the rolling of the fruit between each shot.

	fruit	cam	view	pt_len_1	pt_len_2	pt_cal_1	pt_cal_2	pt_stalk
0	4	right	0	[121, 220]	[115, 31]	[37, 128]	[200, 122]	[118, 186]
1	4	right	1	[137, 222]	[87, 35]	[31, 154]	[191, 111]	[123, 172]
2	4	right	2	[140, 220]	[102, 32]	[34, 151]	[207, 115]	[132, 186]
3	4	right	3	[140, 218]	[104, 33]	[32, 144]	[211, 108]	[127, 171]
4	4	right	4	[131, 31]	[102, 220]	[34, 114]	[200, 140]	[106, 183]
5	4	left	0	[136, 33]	[103, 212]	[38, 100]	[203, 132]	[NaN, NaN]
6	4	left	1	[137, 35]	[90, 218]	[34, 103]	[195, 145]	[NaN, NaN]
7	4	left	2	[161, 31]	[83, 212]	[34, 99]	[198, 171]	[NaN, NaN]
8	4	left	3	[183, 202]	[58, 47]	[47, 180]	[189, 63]	[NaN, NaN]
9	4	left	4	[134, 214]	[95, 31]	[34, 137]	[197, 101]	[98, 50]
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	

Table 6.1: Fruits dataset originally provided.

It is important to note that, since some measurements were taken from an opposite placed camera, some data were appropriately flipped over the x-axis, in order to bring them back into the same reference system of the main camera (arbitrarily chosen).

	fruit	shot	length	caliper	α	$\Delta\alpha$
0	0	0	190	162	1.86	0
1	0	1	194	165	15.4	13.6
2	0	2	192	177	11.8	-3.7
3	0	3	189	183	11.4	-0.3
4	0	4	191	167	-9.18	-20.6
5	1	0	182	168	10.9	0
6	1	1	189	167	14.7	3.8
7	1	2	197	179	23.7	9.0
8	1	3	199	183	-39.2	-62.9
9	1	4	187	167	-12.5	26.7
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 6.2: Cleaned and parsed fruits dataset.

6.4.2 Fruits Data Analysis

In order to generate a data set as realistic as possible, we first proceeded to analyze the data provided, so as to see if there were some types of correlation, especially between the length and caliper of a single fruit, as well as the subsequent α angles of rotation.

Length and Caliper

The first task concerns the best estimate of the length and caliper of a single fruit, as we have 5 different measurements (one for each shot) of them. However, It is quite immediate to think that the most correct thing to do here is to use the maximum of the various measures as the final estimate, as a simple mean will underestimate them.

Once estabilished this, it also must be taken into account, when sampling the values, that the two distributions will probably be correlated. This was confirmed by the Pearson correlation coefficient ρ:

$$\rho(L, C) = \frac{\text{cov}(L, C)}{\sigma_L \sigma_C} = 0.79$$

which suggests a strong correlation and indicates that we cannot sample the values as if they were independent, but we have to sample both together. A visual representation of the underlying data distributions is reported in the first part of Figure 6.12.

Rolling Angle

The analysis of the rolling angles is actually very similar to the previous one, as the angle α_i of the i-th shot is highly correlated with the difference $\Delta\alpha_{i+1}$ of the next shot. In fact, the average Pearson correlation coefficient ρ for $i \in [0, 4]$:

$$\bar{\rho}(\alpha_i, \Delta\alpha_{i+1}) = -0.74$$

that prevents us from sampling the angles as independent, and forces us to to sample all of the 5 angles at the same time. A visual representation of the underlying data distributions is reported in the second part of Figure 6.12.

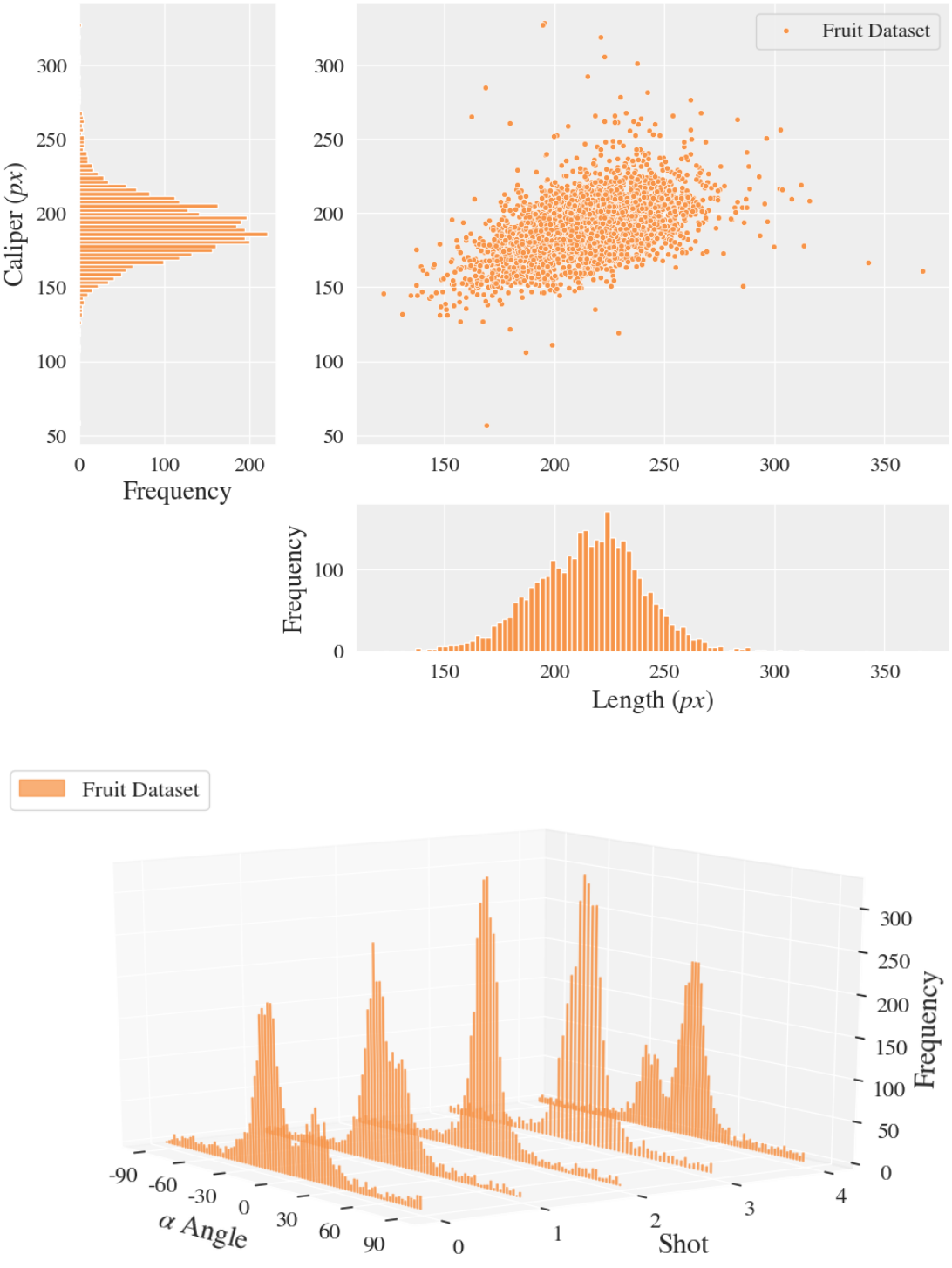


Figure 6.12: Fruits dataset underlying distributions.

Here it is possible to notice the shifting of the rotation angles between the shots, as a consequence of the rolling process. Actually, during this process, there is also a rotation around the major axis of the fruit but, since there are not data available from the real dataset, this will be simulated by generating the angles from a normal distribution.

6.4.3 Fruits Sampling and Defects Generation

Once the properties of the original dataset have been analyzed, it is time to generate the train, validation and test datasets. To do this, a uniform sampling without replacement was carried out on the cleaned and parsed original dataset. As mentioned earlier, we took care to place in the same dataset the pairs of correlated fruit measures, in order to avoid bias. For each fruit sampled, a uniform noise factor of $\pm 5\%$ was then added to both the length and caliper measures and to each α angle independently.

A train dataset of $250k$ samples was then created, together with a validation and test datasets of $5k$ and $25k$ units respectively. An excerpt of these is reported in Table 6.3. Each sampling was carried out on independent splits of the original dataset, with proportion $80\% - 10\% - 10\%$ (which correspond to 2734 fruits for the training set, and 342 for the validation and test sets respectively). The original dataset will actually never be used, neither for validation nor for testing, because of the too low number of examples.

	fruit	shot	length	caliper	α	$\Delta\alpha$	β	$\Delta\beta$
0	0	0	248	215	-5.9	0.0	0.0	0.0
1	0	1	248	215	9.1	15.0	50.8	50.8
2	0	2	248	215	5.1	-4.0	138.4	87.6
3	0	3	248	215	-8.0	-13.1	186.3	47.9
4	0	4	248	215	-6.1	1.9	258.5	72.2
5	1	0	217	194	-2.5	0.0	0.0	0.0
6	1	1	217	194	5.4	7.9	33.5	33.5
7	1	2	217	194	-3.8	-9.2	88.7	55.2
8	1	3	217	194	-4.0	-0.2	121.3	32.6
9	1	4	217	194	16.5	20.5	196.7	75.4
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 6.3: Generated fruits shots dataset.

As said before, for each fruit sampled an angle $\Delta\beta$ of rotation around its major axis is also sampled but, since no information is available from the real dataset, this is generated randomly from a normal distribution $\mathcal{N}(\mu, \sigma)$. With all this information, the dataset accurately describes both the fruit and the rolling process for each shot, and now we only have to compute the various properties of the defects. These, in fact, will be stored in a new dataset in order to speed up the computation of the neural network. For each fruit, therefore, a number of defects ranging from 0 to 5 are projected onto its surface, according to the procedure described in 6.3.2, with an angular position between $[0°, 360°]$ for the longitude and $[-65°, 65°]$ for the latitude. This last range was chosen in order to avoid that some points fall too close to the singularity points, generated by the transformation of polar coordinates used to project the defects on the ellipsoid. The fruit is then rotated for each shot by the angle $\Delta\alpha$ around the z-axis, and by an angle $\Delta\beta$ around its major axis. All the points of the defects are then acquired and the properties computed and stored in a dataset. An excerpt of this is reported in Table 6.4. Here the solidity of each defect is always near 1.00 because the created defects are always convex (but nothing prevents to generate also concave ones).

	fruit	shot	defect_id	pt_cen_x	pt_cen_y	circularity	eccentricity	solidity
0	0	0	0	−52	−39	0.91	0.84	1.00
1	0	0	1	−86	10	0.93	0.90	0.99
2	0	1	0	−48	−37	0.90	0.84	1.00
3	0	1	1	−83	13	0.95	0.91	1.00
4	0	2	0	−64	−36	0.94	0.83	1.00
5	0	2	1	−91	14	0.79	0.84	0.99
6	0	3	0	−57	−42	0.93	0.83	0.99
7	0	3	1	−89	6	0.87	0.87	0.99
8	0	4	0	−49	−49	0.92	0.79	1.00
9	0	4	1	−88	6	0.90	0.87	1.00
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 6.4: Generated defects properties dataset.

Chapter 7

Results and Conclusions

7.1 Software and Hardware Setup

It is now time to describe the model used more in thoroughly, its structure, what it takes as input and what it sends as output, and the training algorithm, in order to make better comparisons later on. A brief description is also given of the hardware setup used, both the one owned and the one provided kindly by Bioretics.

7.1.1 Input Mapping and Training Algorithm

As for the structure, the network has a number of hidden layers fixed at 4, as in the Snake game case, but their sizes are not yet set, since a comparison between different possibilities is made later and it will be reported in the appropriate section.

As for the input, the network continuously compare the properties of two defects, see the process described in Figure 6.7, by accepting a 7-dimensional input vector:

- `input[1-2]`: the difference between the x and y position, already normalized in the interval $[0, 1]$, as described in Section 6.3.1;
- `input[3-5]`: the difference between the three defect properties (circularity, eccentricity, and solidity);
- `input[6-7]`: the progress of the process, measured as the ratio between the analyzed shots (or defects) over the total number of shots (or defects): $\frac{\#shot}{N_{\text{shots}}}$ $\frac{\#defect}{N_{\text{defects}}}$

At the other end instead, the network has two output units:

- one with a sigmoid activation function, used for the binary classification of the two defects into the labels $\{identical, different\}$;
- one without activation function, used to evaluate the value function (then used to train the model, see Section 5.3).

Finally, as already said, we used the REINFORCE algorithm (see Algorithm 16) to train the network, which is a variant of the Actor-Critic algorithm used to solve the Snake game toy-problem. In this case in fact the network can be updated with gradients after having analyzed every fruit, as this is a well defined episode, and it is therefore useless to use the value function also to bootstrap the state-values for each update every few steps, as in the Actor-Critic algorithm. As said before, the network is trained using the Adam optimizer, explained in Chapter 1, with a learning rate of 10^{-4}.

All this written code, from the dataset generation part to the model training process, was written in the Python language. Several libraries have been used, among them TensorFlow, for the deep learning part, and Numpy, Scipy, and Pandas, for the dataset analysis and the fruits generation parts.

7.1.2 Hardware Specifications and Time Required

All the hardware used in the project consists of only two machines, one personally owned and one provided kindly by Bioretics. The first one is a simple personal computer with a GNU/Linux Ubuntu 18.04.4 LTS distribution, which runs on an Intel Core i7-3610QM 2.30 GHz quad-core with 8 GB RAM as the CPU and a NVIDIA GeForce GT 650M as the GPU. The second one is a server with GNU/Linux Ubuntu 16.04.6 LTS distribution, which runs on an Intel Core i7-7700 3.60 GHz quad-core with 32 GB RAM as the CPU and a NVIDIA GeForce GTX 1080 as the GPU.

The parts of software that have taken the longest time to run are undoubtedly the generation of the datasets used and the corresponding training part. The first part took about 3 days in total to compute in multiprocessing on the CPUs of the two computers, for a total of more than 10^6 fruits generated. For the training part, the single process took less time, about 16 hours, but, with more simulations to do, the total amount of time exceeded 4 days of computation on the available GPUs.

7.2 Applicability Study

In order to effectively study the applicability of the model, different tests were conducted under different conditions. In particular, we have tested different ways of rolling around the main axis, in order to have as complete a view as possible on the application possibilities of the model. In addition, we made a comparison with two alternative models, a neural network with wider hidden layers and one trained using supervised learning.

For the sake of simplicity, we started with only a qualitative comparison between the various cases, while the numerical results are given later in this section. All the graphs of the training processes are then presented first, together with visual examples of the simulated shots acquired for each rolling case, and a brief comment on the comparisons made. After this, all the numerical results are presented, even with the help of some graphics, and a final commentary on the overall study is provided.

As already said, all the results are expressed in terms of accuracy of guessing, obtained by rescaling in the interval $[0, 1]$ the mean of the average rewards for each fruit. In order to avoid overfitting the early stopping method, explained in Chapter 1, has been used. Therefore, the model is validated every $1k$ steps, and the best one is saved for testing.

7.2.1 Comparison Between Different Rolling Processes

As described before, the real dataset allows us to estimate the angle of rotation for each fruit around the z-axis, indicated by α, but not around their major axis, indicated by β. The initial value of the β angle in the starting shot was then arbitrarily set to $\beta_0 = 0$, since this is not fundamental for the dynamics (being the ellipsoid symmetric), and the next angle differences between shots $\Delta\beta_i$, for $i \in [1, 4]$, were sampled from several different normal distribution $\mathcal{N}(\mu, \sigma)$, in order to make some comparisons.

Four distributions were studied: $\mathcal{N}(0, 0)$, $\mathcal{N}(0, 15)$, $\mathcal{N}(30, 15)$, $\mathcal{N}(60, 15)$, which correspond to four different dynamics: no rolling, random equiprobable rolling, small directional rolling and large directional rolling. For all of these a neural network with shape $42 - 48 - 36 - 18$ is used, a little bigger than the one used to solve the Snake game toy-problem, with each of its hidden layers with $\tanh(x)$ as the activation function.

No Rolling

The first case studied is the trivial case, in which the angles $\Delta\beta_i$ are all zero. In this case the fruit does not rotate around its major axis, but only around the z-axis, following the dynamic generated previously. An example of the simulated shots featuring no rolling is represented in Figure 7.1, while the graphical results are reported in Figure 7.2.

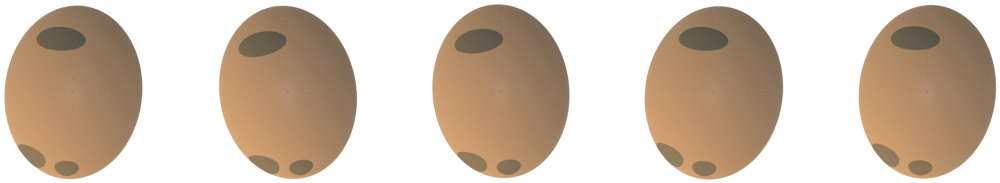


Figure 7.1: Five consecutive simulated shots in case of no rolling.

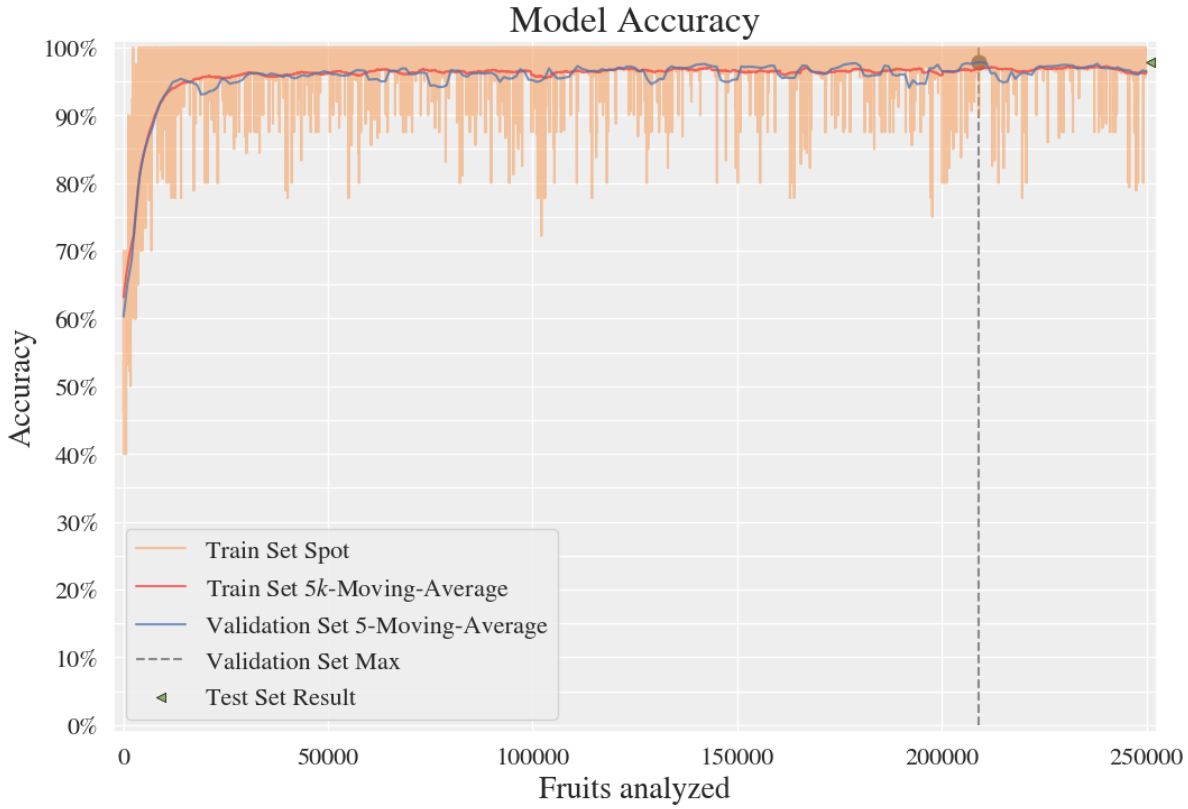


Figure 7.2: Results in case of no rolling.

Randomly Equiprobable Rolling

In the second case analyzed, the angles $\Delta\beta_i$ follow the normal distribution $\mathcal{N}(0, 15)$. The fruits therefore rotate both around the z-axis and around their major axis. An example of the simulated shots featuring equiprobable rolling is represented in Figure 7.3, while the graphical results are reported in Figure 7.4.

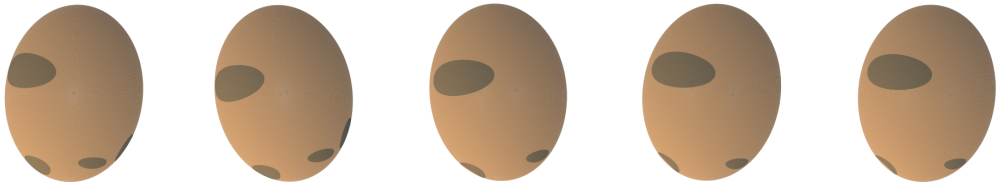


Figure 7.3: Five consecutive simulated shots in case of equiprobable rolling.

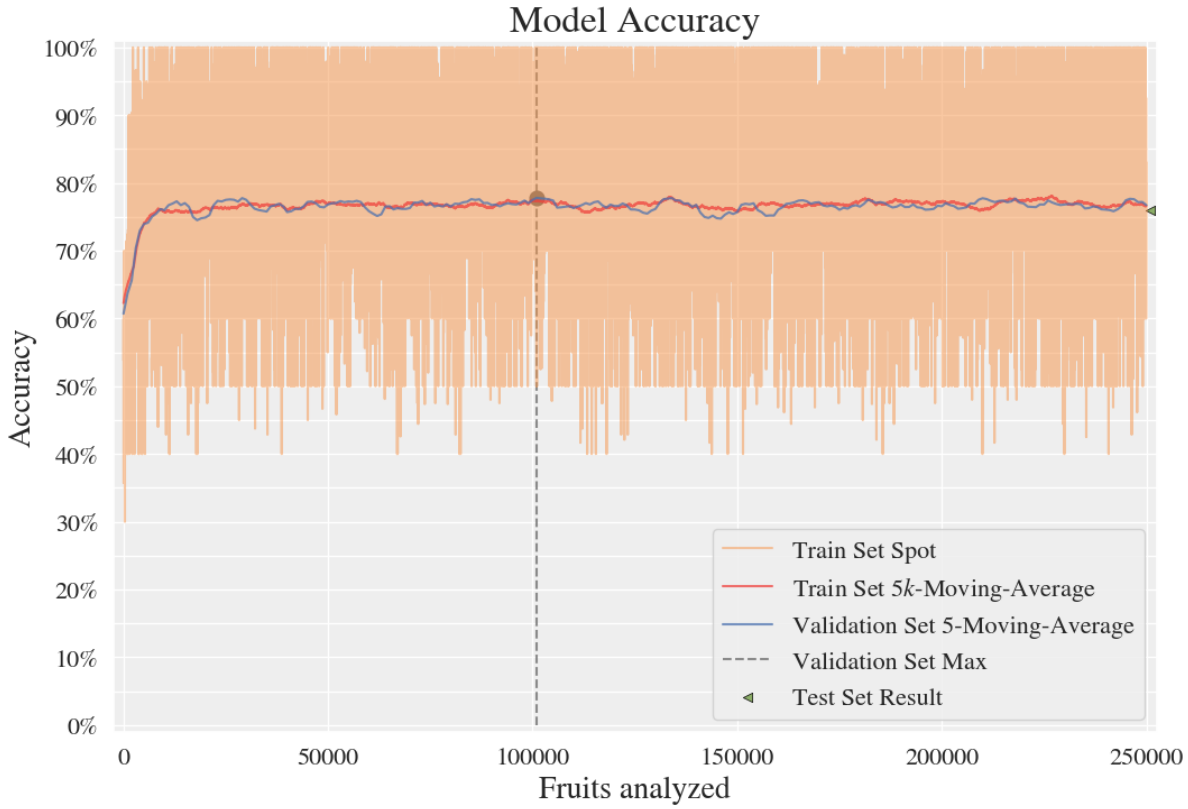


Figure 7.4: Results in case of randomly equiprobable rolling.

Small Directional Rolling

In the third cases studied, the fruit rotates as before, but the angles $\Delta\beta_i$ now follow the normal distribution $\mathcal{N}(30, 15)$ in order to simulate a rolling process with a fixed direction. An example of the simulated shots featuring this small directional rolling is represented in Figure 7.5, while the graphical results are reported in Figure 7.6.

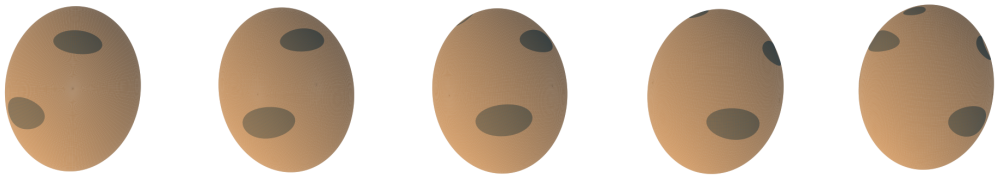


Figure 7.5: Five consecutive simulated shots in case of small directional rolling.

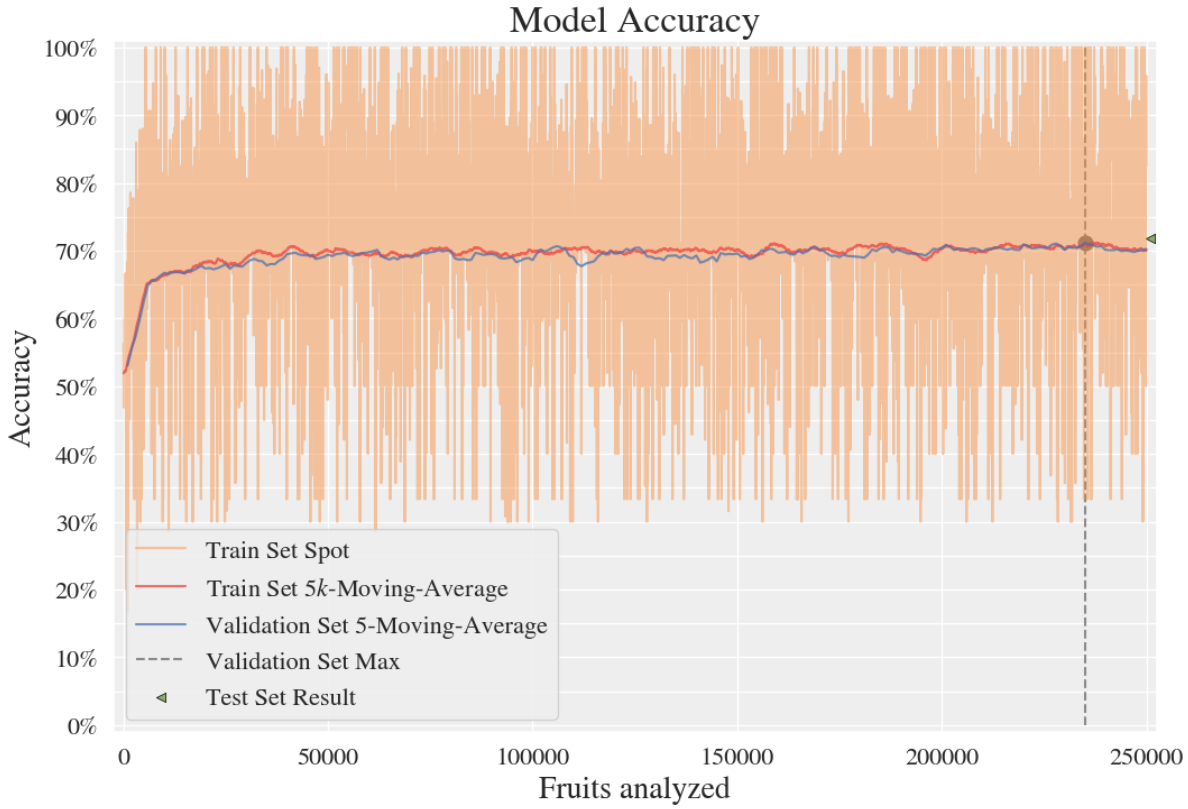


Figure 7.6: Results in case of small directional rolling.

Large Directional Rolling

In the last rolling case analyzed, the angles $\Delta\beta_i$ follow the normal distribution $\mathcal{N}(60, 15)$, an almost extreme case in in which the fruit takes an average of almost a full turn on itself. An example of the simulated shots featuring this small directional rolling is represented in Figure 7.7, while the graphical results are reported in Figure 7.8.

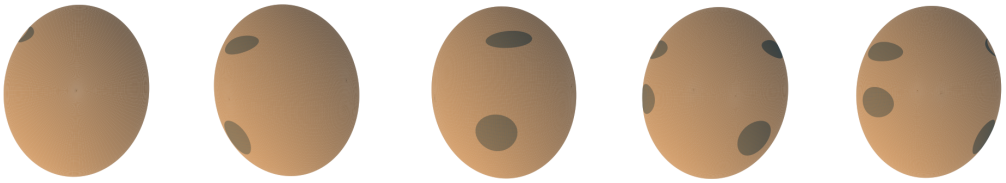


Figure 7.7: Five consecutive simulated shots in case of large directional rolling.

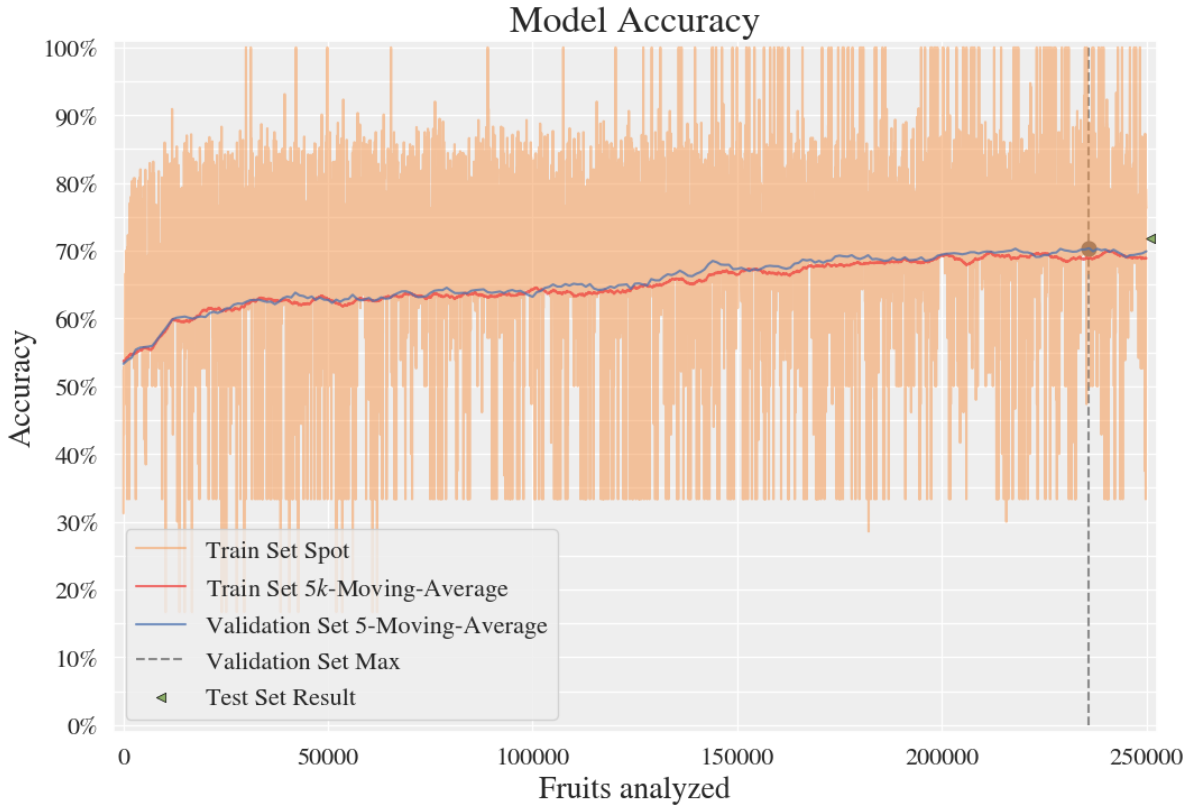


Figure 7.8: Results in case of large directional rolling.

Comments on the Comparison

For ease of comparison, both now with regard to the various rolling dynamics and shortly with the other models and types of learning, the overall trend of the four cases in the figure is shown in Figure 7.9, where only the results of the validation and test set are represented, as the train set has no statistical significance.

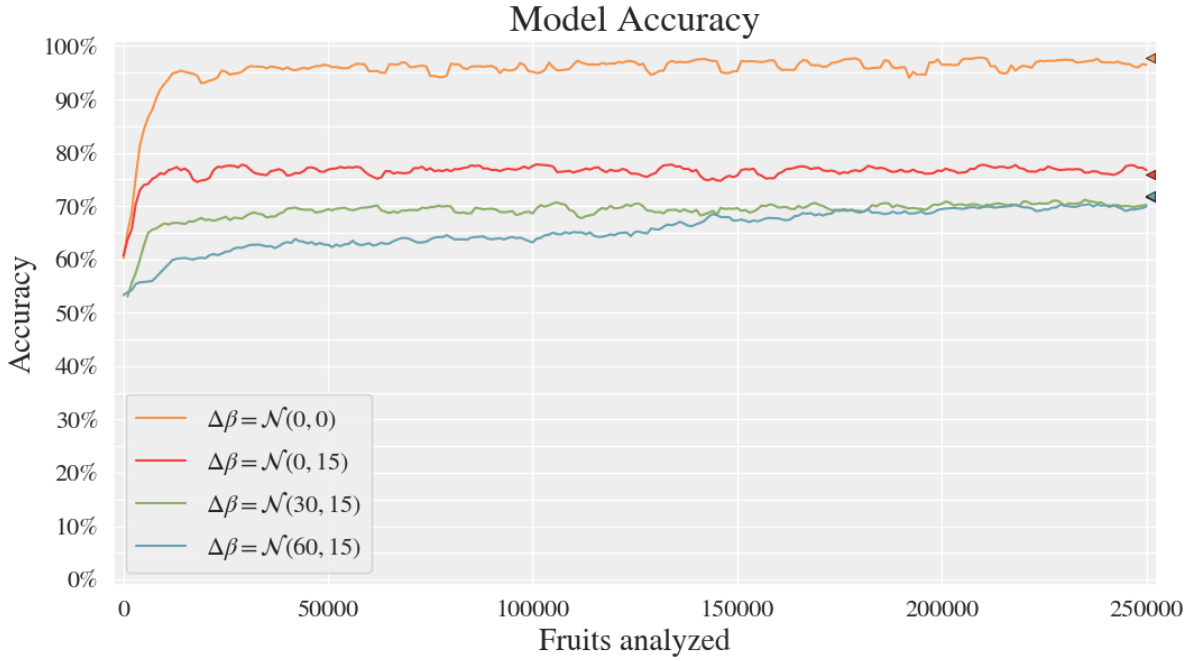


Figure 7.9: Results for several rolling cases.

It is immediate to notice, from the graphs of the training processes, how the training and validation trends are almost identical. This is a sign that the model generalizes well, and that the validation dataset is very similar to the training one. As a consequence of that, we can conclude that the projection of the defects on the surface of the fruit is more relevant than the size of the fruit (or its rolling dynamic), since the difference between the train, validation, and test datasets lies only in the latter.

Regarding the accuracy of the model, in the first case it is extremely high, undoubtedly due to the fact that, although there is a rotation around one axis, the other one is completely ignored. As one might then expect, this precision significantly decrease with the introduction of the rotation around the major axis. Despite being small in fact, the

randomness introduced in the second case lowers the score by $\sim 20\%$. This is however still an interesting results, as the underlying dynamic is no longer trivial.

The accuracy is then further reduced when the rolling becomes directional, as in the third and fourth case. In particular a drop in score from the previous case of $\sim 5\%$ is measured. However, the increase of the rolling angles doesn't seem to affect too much the precision. There's not in fact a big difference between the third and the fourth case, a sign that we've probably reached the lower limit of the model's accuracy. The only difference we may notice here is in fact the speed of convergence to the final value, as the steepness of the training and validation trends is more gentle in the last case.

7.2.2 Comparison Between Different Models

To make the study as complete as possible, it is important to compare the model used with some alternatives. For this we went to compare the original neural network with a version with wider layers and with a version trained using supervised learning, on all four rolling cases described above. As before, only short descriptions of the comparisons are given here, together with the results of the validation and the test set and a short comment, while later in the next section are reported the numerical results.

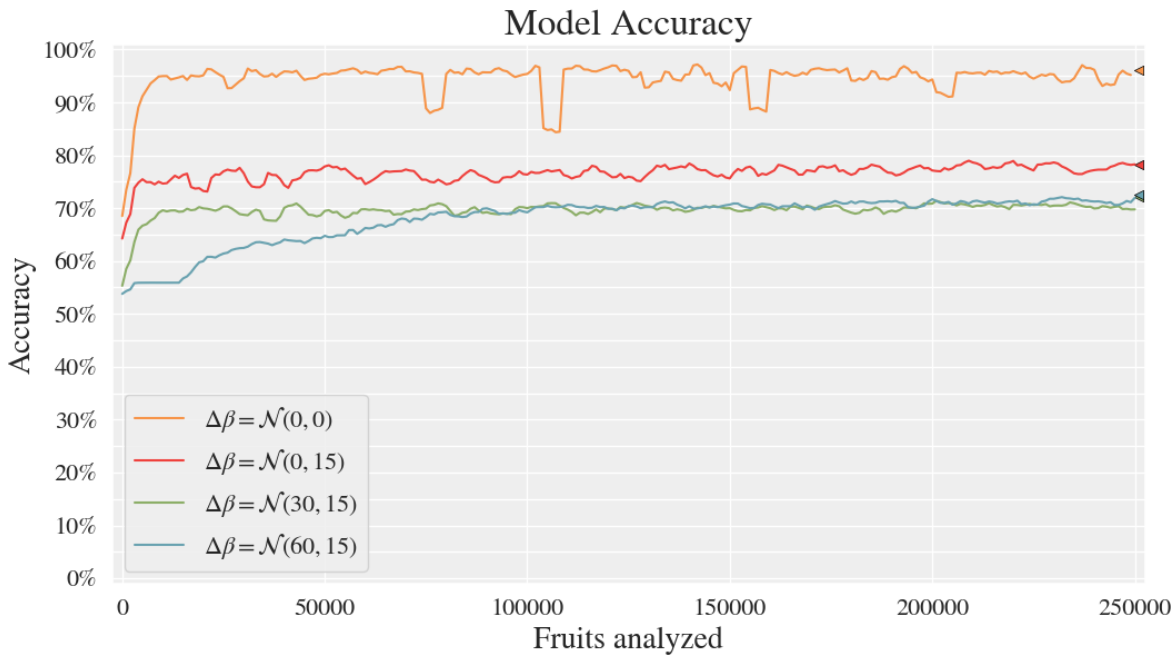
Wider Neural Network Supervised Learning

First of all, the original model used was compared with a wider neural network, of the form $126 - 144 - 108 - 54$, three times larger than the original one. Besides that, the input and output structure is kept the same, as well as the layers activation functions. The graphical results are reported in Figure 7.10(a).

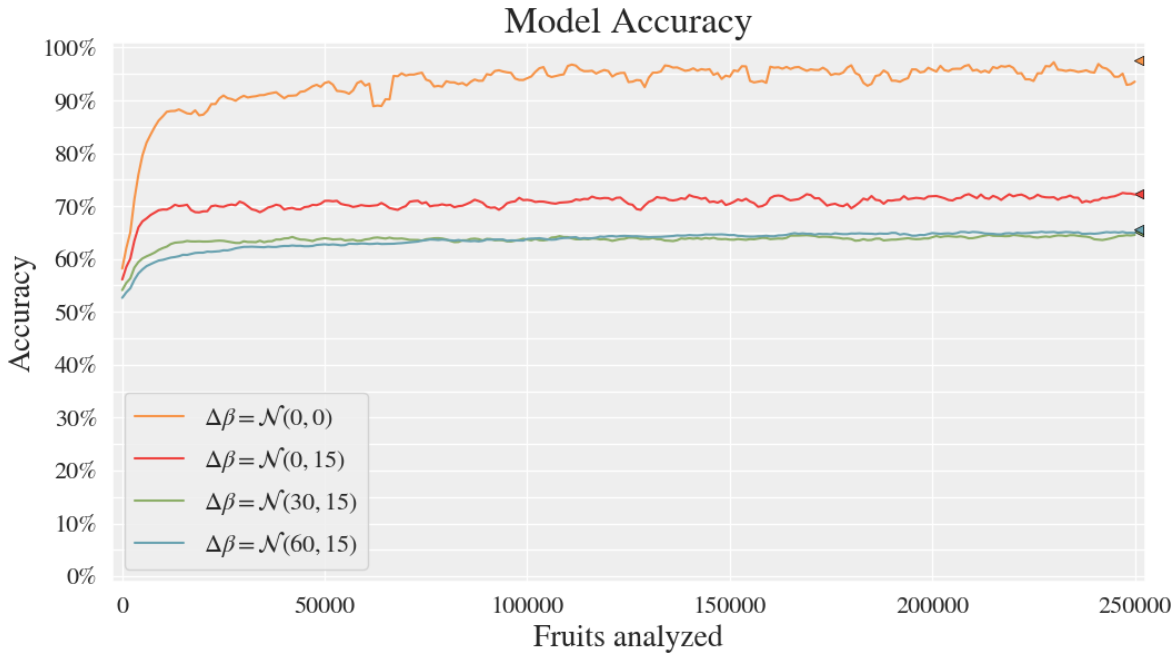
Subsequently, the model was then compared with a network of the same size but trained using supervised learning. Here the output is therefore reduced to the classification unit only (the one with the sigmoid as the activation function) and the loss function used is the binary cross entropy. The graphical results are reported in Figure 7.10(b).

Comments on the Comparison

In the first graph, it is immediate to notice the strange trend of the model for the no rolling case. The depicted negative peaks are peculiar in case of overcapacity, i.e. they indicate that the model in question has probably too many unnecessary parameters to



(a) Wider network.



(b) Supervised learning.

Figure 7.10: Results for several rolling cases with different models.

solve the task. In this case, the shots of the fruit, which does not rotate around its major axis, are probably a problem "too simple" to solve, and are causing some instability in the network. Apart from that, the final results and trends of the other cases do not seem to be so different from those presented previously.

As for the model trained with supervised learning, we can see that this gives rise to training curves very similar to those seen previously. Despite this the final results seem to be slightly lower than those measured with reinforcement learning.

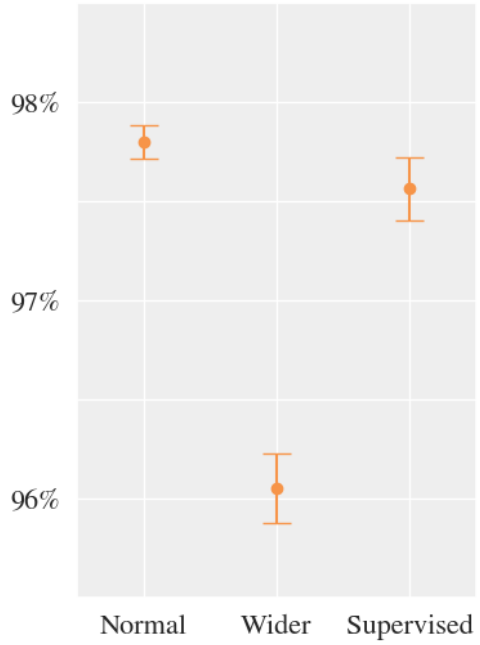
7.2.3 Numerical Results

To draw the most objective conclusions, the numerical results of the accuracy measured of the various models are given in Table 7.1. Actually the final score is calculated by splitting the test set into 5 independent parts (therefore each composed of $5k$ fruits), and averaging the measured accuracy obtained on each of them.

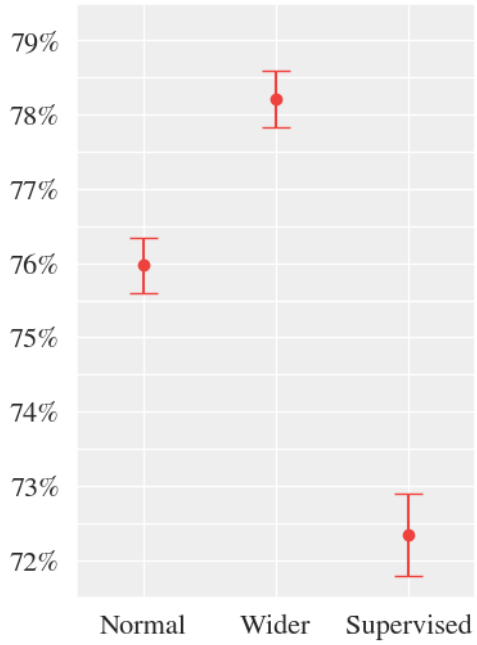
$\Delta\beta$ distribution	Network size	Learning Type	Accuracy $\pm$ SD
$\mathcal{N}(0,0)$	Normal	Reinforcement	97.8 ± 0.1
$\mathcal{N}(0,15)$	"	"	76.0 ± 0.4
$\mathcal{N}(30,15)$	"	"	71.8 ± 0.5
$\mathcal{N}(60,15)$	"	"	71.9 ± 0.3
$\mathcal{N}(0,0)$	Wider	Reinforcement	96.1 ± 0.2
$\mathcal{N}(0,15)$	"	"	78.2 ± 0.4
$\mathcal{N}(30,15)$	"	"	72.2 ± 0.5
$\mathcal{N}(60,15)$	"	"	72.6 ± 0.2
$\mathcal{N}(0,0)$	Normal	Supervised	97.6 ± 0.2
$\mathcal{N}(0,15)$	"	"	72.3 ± 0.6
$\mathcal{N}(30,15)$	"	"	65.2 ± 0.3
$\mathcal{N}(60,15)$	"	"	65.6 ± 0.1

Table 7.1: Models accuracy comparison table.

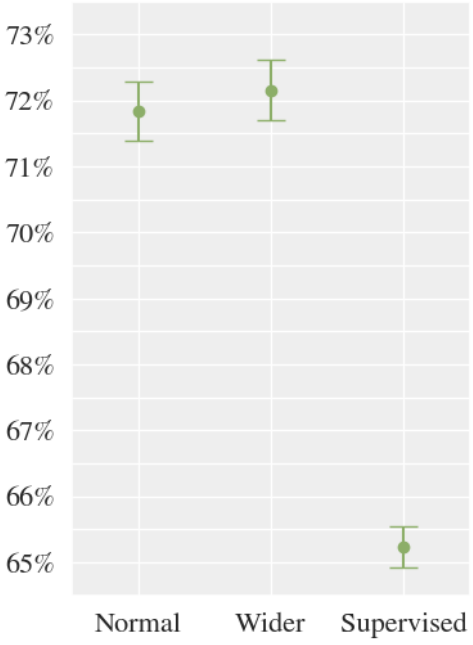
A graphic representation of all the results is reported in Figure 7.11, where they have been divided by rolling cases instead. The error bars, fixed at 3σ, have been represented here, so that it is easier to comment on the comparability between the various models.



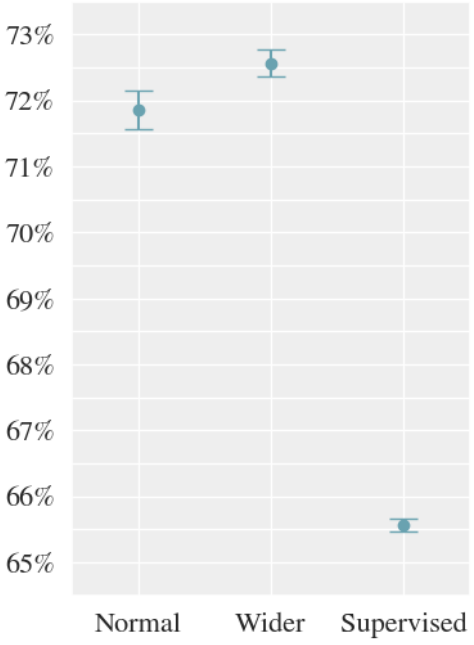
(a) $\mathcal{N}(0, 0)$



(b) $\mathcal{N}(0, 15)$



(c) $\mathcal{N}(30, 15)$



(d) $\mathcal{N}(60, 15)$

Figure 7.11: Models accuracy comparison.

Let's start with a case-by-case model analysis of the accuracy trend as the degree of rolling increases. For the standard case, with a normal size network and using the reinforcement learning algorithm, we can see how the accuracy starts with a very high value in the case of no rolling, and then drops as the rolling introduced increases. In the case of directional rolling, however, the value seems to settle around $71.8 \sim 71.9$, with an irrelevant difference between the two cases as the two are perfectly comparable. This same trend, including the compatibility of results for directional rolling, is also repeated in the case of a wider network and in the case of supervised learning.

We now move on to analyze the trend of the models for each rolling dynamics studied. For all of them, we can see that the model with the wider network is generally better performing, except in the no rolling case, where we have already said that this is probably due to an instability of the model itself caused by the overcapacity. However, the differences with the original model are minimal, often only reduced to a few points.

The interesting thing that we can see from these analyses is instead that the models trained through reinforcement learning are generally more accurate than those trained through supervised learning, as it's clearly visibile from Figure 7.12 (only the mean value of the accuracies measured is reported here).

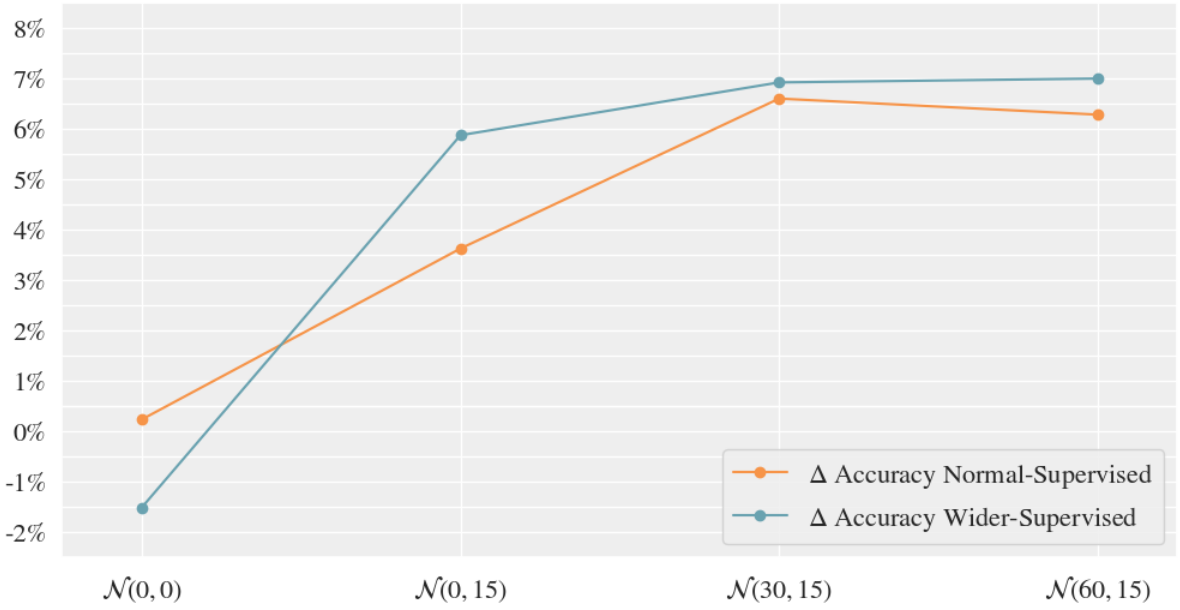


Figure 7.12: Difference between accuracy of differently trained models.

This is particularly relevant, and may lead us to think that reinforcement learning is to be preferred if some of the input encodes information in a time sequence (like the overall process of analysis of a single fruit in this case). This might also be supported by noticing the increase in the difference of performance between the models, when there is a rolling process compared to the no rolling case, which increases even more when the rolling becomes directional.

7.3 Final Considerations

During this thesis work, the theory and models typical of machine learning and deep learning were studied in depth. From this, we then went on to study the reinforcement learning approach, starting from its theoretical basis up to the implementation algorithms in a deep learning perspective through neural networks. A toy problem, the Snake game, was then solved with these, in order to test the correct implementation of policy gradient methods in a deep learning perspective.

Things have become more complicated in moving from this to an industrial problem: the recognition and connection of defects on the same fruit, starting from images acquired at different time steps. It was therefore necessary to find a way to model the problem effectively, in order to be able to adapt and apply the previously developed algorithm. We then started by simulating the fruits rolling process and shots acquisition, and then moved on to the synthetic generation of the fruits and of the defects present on them. Starting from real data, a custom dataset was then created with which a study of the applicability of the model was made. In this, we went to compare different fruits rolling dynamics and the effectiveness of different learning types.

After all the comments that have been made in this chapter on the various cases studied, one thing seems quite clear: the consistency of reinforcement learning methods also for industrial problems. That is precisely the point we would like to make from the beginning: although its methods and algorithms find their natural development in (video)games, as explained in the previous chapter, this does not preclude the formulation of a problem of any kind in a reinforcement learning perspective. In addition to this, we've seen from the various cases studied that it can actually be very effective and how it is perfectly comparable, if not superior, to other types of learning.

Future Developments

This thesis work obviously cannot be considered a complete solution to the problem of fruit defects, already extensively described, as many refinements on the code need to be done and tests to be conducted. These have not been done yet because of obvious difficulties, that mainly involve time required and workload.

First of all, it would be necessary to increase the number of train examples, as well as validation and test ones, since deep learning works much better with datasets of orders of magnitude higher than those used in this project. Together with this it should then be carried out a process of cleaning and optimization of the code used, in order to speed up the processes and remove all those bugs that surely are still there.

Secondly, it would be necessary to improve the creation of the fruits and defects, both in terms of their shape and size and in terms of the process of surface projection and shots acquisition simulation. Although these have proved to be effective in the study of the problem addressed, this is only a starting point, as it should be possible to simulate several shapes and improve the quality of the projection, of both defects and fruits.

Finally, it would be necessary to do several other tests on different models, perhaps even with including more complex architectures such as recursive neural networks. This would allow even more to exploit the effectiveness of reinforcement learning in managing input presented in a time sequence, and not to mention all the fine tuning work of the network and algorithms hyperparameters.

Bibliography

[Barnard, 1993] Barnard, E. (1993). Temporal-Difference Methods and Markov Models. *IEEE Transactions on Systems, Man, and Cybernetics*.

[Barron, 1993] Barron, A. R. (1993). Universal Approximation Bounds for Superpositions of a Sigmoidal Function. *IEEE Transactions on Information Theory*.

[Bishop, 2006] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

[Cybenko, 1989] Cybenko, G. (1989). Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals, and Systems*.

[Duchi et al., 2011] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning*.

[Goodfellow et al., 2017] Goodfellow, I., Bengio, Y., and Courville, A. (2017). *Deep Learning*. MIT Press.

[Hammond, 2017] Hammond, M. (2017). Deep Reinforcement Learning in the Enterprise: Bridging the Gap from Games to Industry. In *Artificial Intelligence Conference Presentation*.

[Hinton, 2012] Hinton, G. E. (2012). Slides from the course in *Introduction to Neural Networks and Machine Learning*. `https://www.cs.toronto.edu/~tijmen/csc321/`.

[Hornik, 1989] Hornik, K. (1989). Multilayer Feedforward Networks are Universal Approximators. *Neural Networks*.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. L. (2014). Adam: A method for stochastic optimization. *Conference paper at ICLR 2015*.

[Mitchell, 1997] Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.

[Mnih et al., 2016] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous Methods for Deep Reinforcement Learning. In *Proceedings of The 33rd International Conference on Machine Learning*.

[Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602*.

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

[Ramachandran et al., 2017] Ramachandran, P., Zoph, B., and Le, Q. V. (2017). Swish: A Self-Gated Activation Function. *ArXiv e-prints*.

[Rojas, 1996] Rojas, R. (1996). *Neural Networks: A Systematic Introduction*. Springer.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning Representations by Back-Propagating Errors. *Nature*.

[Silver, 2015] Silver, D. (2015). Slides from the course in *Reinforcement Learning*. `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`.

[Silver et al., 2016] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*.

[Silver et al., 2017] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui,

F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. (2017). Mastering the Game of Go without Human Knowledge. *Nature*.

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning - An Introduction*. MIT Press, 2 edition.

[Szepesvari, 2010] Szepesvari, C. (2010). *Algorithms for Reinforcement Learning*. Morgan & Claypool.

[Vinyals et al., 2019] Vinyals, O., Babuschkin, I., Czarnecki, W., Mathieu, M., Dudzik, A., Chung, J., Choi, D., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J., Jaderberg, M., and Silver, D. (2019). Grandmaster Level in StarCraft II Using Multi-Agent Reinforcement Learning. *Nature*.

[Williams, 1992] Williams, R. J. (1992). Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*.