Trabajo Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Application of neural networks to the collision avoidance problem in 2D based on the TensorFlow library

Autora: Rebeca Fernández Niederacher

Tutores: Jesús Iván Maza Alcañiz, Aníbal Ollero Baturone

Departamento de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Seville, 2018

Trabajo Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Application of neural networks to the collision avoidance problem in 2D based on the TensorFlow library

Autora:

Rebeca Fernández Niederacher

Tutores:

Jesús Iván Maza Alcañiz

Profesor Titular de Universidad

Aníbal Ollero Baturone

Catedrático de Universidad

Departamento de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018

Trabajo Fin de Grado: Application of neural networks to the collision avoidance problem in 2D based on the TensorFlow library

Autora: Rebeca Fernández Niederacher
Tutores: Jesús Iván Maza Alcañiz, Aníbal Ollero Baturone

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

To my family,
my aunt Marlies,
my friends

Acknowledgements

I would like to acknowledge, firstly, my supervisor Jesús Iván Maza Alcañiz who gave me the opportunity to start learning an unknowledge area for me, guiding throughout this project.

In addition, I would like to express my gratitude to Jose Andrés, member of the investigation group, who give me his unconditional help. Thanks to teach me your knowledges and make me part of your research.

Thanks to my family for showing interest in everything I have done and giving me their support and understanding. Specially, thanks to my father, he always wished me good luck before going to the ETSI, to my mother who trusts blindly in me and to my aunt Marlies, for opening my eyes and teaching me that every minute counts.

Lastly, I would like to thank also my classmates, they have been the best support in the worst and the best participants in the good moments. Thanks also to my rugby team for giving me the strength to keep going, specially to my best friend and mate Inés who always believed in me even when I did not.

Rebeca Fernández Niederacher
Student of Robotics, Electronics and Mechatronics engineering

Seville, 2018

Agradecimientos

Me gustaría, antes de todo, agradecer a mi tutor Jesús Iván Maza Alcañiz la oportunidad de adentrarme en un área totalmente desconocida para mí, apoyandome en todos estos meses.

A su vez, agradecerle a Jose Andrés, miembro del grupo de investigación, su incondicional ayuda. Gracias por enseñarme todos tus conomientos y hacerme participe de una parte de la investigación que llevas a cabo.

Gracias a mi familia por interesarse siempre por lo que hago y prestarme siempre su apoyo y comprensión. En especial, a mi padre por desearme suerte todos los días antes de ir a la ETSI, a mi madre por confiar a ciegas en mí y a mi tia Marlies, por abrirme los ojos y enseñarme que cada minuto cuenta.

Por último, agradecer, por un lado, a mis compañeros de clase por ser el mejor apoyo en los malos momentos y grandes participes de los buenos. Por otro lado, a mi equipo de rugby por darme la fuerza necesaria para seguir adelante, pero sobretodo, a mi gran amiga y compañera Inés que siempre ha confiado en mí cuando ni yo misma lo hacía.

Rebeca Fernández Niederacher
Estudiante de Ingeniería Electrónica, Robótica y Mecatrónica

Sevilla, 2018

Abstract

The objective of this paper is the learning and initiation into the world of neural networks using the Google tool, TensorFlow. In order to do this, we consider a series of algorithms whose purpose is the control of a drone which can move in a specific environment, avoiding static and mobile obstacle while, at the same time, guaranteeing a safe navigation. This tool is the main different with respect to older research in this field. Furthermore, we look into the structure and the diverse tools that this platform offers with the intention of discovering the areas in which TensorFlow can be useful. Therefore, the organization of this paper is structured as follows:

In the first place, we offer an introduction that covers Neural Networks that are so important nowadays in the wide range of application available. We also explain what they are based on and how the information is used.

Next, TensorFlow structure is briefly introduced, explaining also how it works and some of the basics tools provided by it.

After that, the setting in which we are currently working is illustrated in three different steps. First, a data set is created, then the TensorFlow algorithms are implemented for the different scenarios and finally the "learning" obtain by the neural networks are analysed.

Lastly in our conclusions we offer two significant points: first, we demonstrate the findings of the different neural networks in the simulator provided and, second, the conclusions that we have reached in this paper and the future line of researches that this study put forth.

Abbreviated index

Contents

Abbreviations

UAV	Unmanned aerial vehicle
NN	Neural network
AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
TF	TensorFlow
CNN	Convolutional neuronal network
ReLu	Rectifier Unit Activation function
Tanh	Hyperbolic Tangent function
RNN	Recurrent Neural Network
FL	Fully Connected architecture
WP	Waypoint
UAL	UAV Abstraction Layer

Introduction

"Drones overall will be more impactful than I think people recognize, in positive ways to help society".

BILL GATES

To begin with, it is important to know in which context the investigation is situated and how the idea materialise.

Unmanned aerial vehicle (UAV)

Since its invention, UAVs or drones, how they are popularly called, have had an exponential rise and development, becoming more importance day by day. Their functionality is astonishingly widespread as they can be employed for many different tasks and applications, from the supervision of a geographical area to the transport of a package.

There are several reasons why drones are so important and practical in so many circumstances. On one hand, it is no longer necessary to use a human in order to pilot an zone with an air plane, and thus, avoiding the danger that could arise in a military mission or in the search of people in areas of difficult access.

On the other hand, the cost of using an unmanned aircraft is much lower compared to using a commercial aeroplane and therefore we can use drones in more applications.

Their small size and their manoeuvrability provide a lot of options and even more if a system with more than one UAVs is implemented.

In order to have an idea of how useful they are, some of the tasks that are carried out with drones are shown:

- the transport of a small package from one place to another without the need of a delivery person. This task is used in parcel companies like DHL ;

Source:
https://science.howstuffworks.com/transport/flight/modern/10-non-murderous-drones.htm

Figure 1.1 A drone for package delivery.

- a system with a few drones synchronized together to transport a heavy load, making the process faster and cheaper;

Source:
https://novologisticablog.blogspot.com/2015/07/

Figure 1.2 Quadrotor transporting a heavy load.

- the aforementioned synchronized drones are employed to supervise and monitor the state of an open field or the location of endangered species whose position is relevant. Also, drones give firefighters an accurate information regarding the behaviour of a fire in forests or they can even prevent them .

Source:
https://contactohoy.com.mx/disenan-un-sistema-de-deteccion-automatica-de-incendios-para-drones/

Figure 1.3 Quadrotor in a fire.

Hereafter, the terms drone, robot or UAV are going to be used indistinctly.

Neural Networks for UAVs

There are many researches about the usefulness of a neural network (NN) controlling an UAV. As known, it is not easy to move an UAV from one position to another without collisions in specific environments. Here is where NNs are getting good results and therefore can be a good option to control UAVs. The algorithms have to satisfy the safety control of the UAVs while, at the same time, trying to lower computational cost.

In a previous project, these algorithms were calculated using Matlab Toolbox for Neural Network and a simulator provided from the Department. However, a new idea of implementation using TensorFlow instead of it emerged. TensorFlow, which will be introduced in Section 3, is an open source software library for high performance, numerical computation, and its main objective is to make work easier of the program engineer. Therefore, the aim of this paper is to create algorithms with the help of TF, capable of "teaching" drones in real time and in specific scenarios and then, analyse the different results using the same simulator. We can consider this task as a new black box between inputs and outputs, due to these values are the same for both algorithms. The difference here is how they compute the new outputs.

Motivation

This project has come into being through research into Deep Learning methods to create a software for unmanned aircraft with the help of TensorFlow.

Some of my personal motivations during this project have been to understand, implement and be able to explain what concepts such as Artificial Intelligence (AI) and neural networks (NN) are, and how to use TensorFlow in many different applications. At present, these terms are used in a large number of areas, receiving a lot of attention in a short period time. In addition, it has also been necessary to use ROS, Gazebo and UAL in this research, which are also interesting and potential tools in this days. After finishing this project, I will be able to use some of the NNs applied nowadays in AI and how they can be exploited to implement the guiding of UAVs by using TensorFlow. Therefore and to conclude, this research gives me the possibility to investigate areas which were unknown for me, becoming a good starting point for many research fields.

Objective

The main objective of the project is to see if TensorFlow can be considered a practical tool to create neural networks for different applications and, more specifically, to "teach" UAVs how to move in different scenarios.

Because of the complexity and extensive area of investigation research covered in this type of project, it is necessary to review the basic concepts of it. After acquiring some important ideas, the software is implemented, and the results are shown in a simulation.

The objectives that will be covered are the following:

- introduction to neural networks;
- TensorFlow and TensorBoard;
- dataset extraction;
- training of the scenarios;
- results tested in a Gazebo simulator;
- conclusions and future investigations.

Structure of the project

The structure of the project follows the aforementioned objectives explained previously. We start with an introduction on the topic of neural networks and how it works for different Machine and Deep Learning tasks. Once the main concepts are understood, a brief explanation about what TensorFlow and Tensorboard are and how to work with them, providing examples. Then, the dataset extraction and the neural networks computed with TensorFlow are explained for the different scenarios. At the end of this project, the algorithms will be tested in an UAVs simulator and the results discussed, giving some conclusions about the performances.

Artificial Neural Networks

I think the brain is essentially a computer and consciousness is like a computer program. It will cease to run when the computer is turned off. Theoretically, it could be re-created on a neural network, but that would be very difficult, as it would require all one's memories.

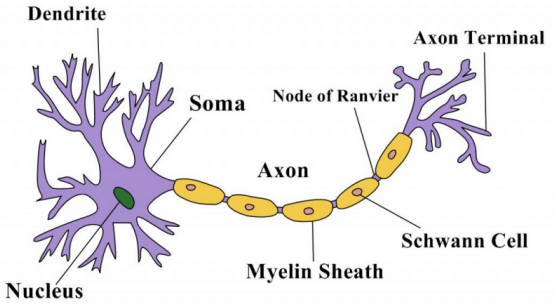
STEPHEN HAWKING

To start with and to understand better what is an Artificial Neural Network, it is necessary to know what a human neuron is and how it works.

Introduction. Brain neuron

A brain neuron is composed mainly by dendrites, soma or cell body and an axon. Dendrites are the input channel and they pick up information from the neuron and send it to the soma. Then the soma processes this information, by sending a new signal to the next neuron through an output channel called an axon. Brain neurons are parallelly interconnected and their information is transferred by electrical impulses.

The following Figure 2.1 is an illustration of a human neuron with its different parts:



Source:
https://owlcation.com/stem/Structure-of-a-Neuron

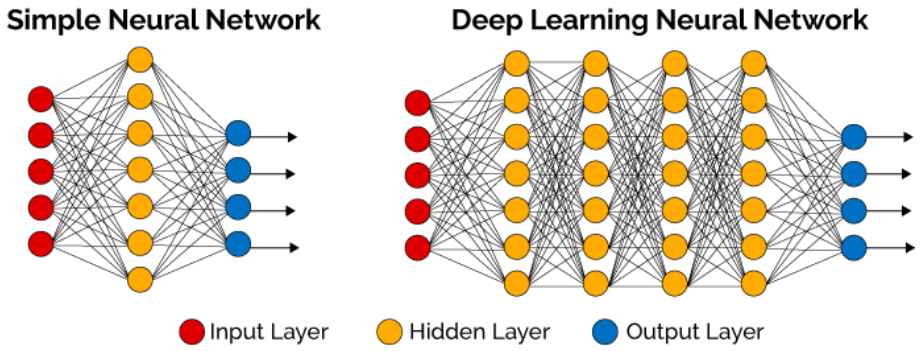
Figure 2.1 Basic structure of a neuron.

Artificial Neural Network

Artificial Neural Network (ANN) is a type of algorithm which can model huge and complex problems, like Machine (ML) and Deep Learning (DL) tasks, in an effective and efficient way. These are concepts, inside

Artificial Intelligence focused on the ability of machines to receive a set of data and learn and adapt it for themselves. There are two types of ML tasks: supervised and unsupervised. In the first one, there are input and output variables and we use an algorithm to learn a mapping function from the input to the output. In the second one, there is only input data and the goal is to model the underlying structure or distribution of the data to learn more about it. In this paper, we will focus in supervised tasks. DL goes one step further than ML and tries to adapt computers with the ability to reason like humans. In this context, learning consists of identifying complex patterns in a wide range of data.

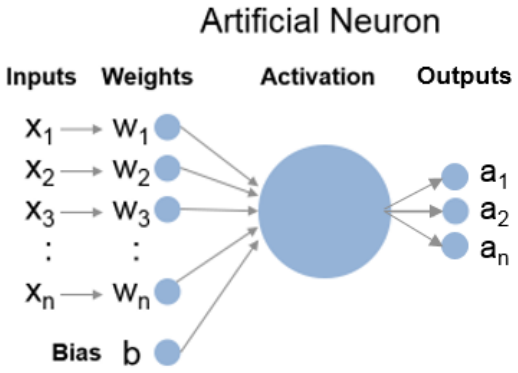
An ANN operates similar to the human brain and is based on a collection of connected nodes with specific weights and is typically organized in layers. Each layer can perform different kinds of transformations on their inputs. Artificial Neural Networks with only one or two hidden layers, are less complex but require more information regarding features. If there are more than one or two layers, as shown in the right picture of Figure 2.2, it is called Deep Learning Neural Network. The more layers the model has, the deeper it is.



Source:
https://www.xenonstack.com/blog/log-analytics-with-deep-learning-and-machine-learning

Figure 2.2 Simple Neural Network compare to Deep Learning Neural Network.

When a node receives an input with a weight, it changes its internal state or its activation function according to that input and produces an output. These values are modified by the learning process, which is governed by a learning rule. Normally, a threshold or bias is implemented which provides every node with a trainable constant value. The next image shows a single node with its inputs, outputs, weights, bias and activation function.



Source:
https://blogs.mathworks.com/loren/2015/08/04/artificial-neural-networks-for-beginners/

Figure 2.3 Artificial Neural Network representation.

An example is demonstrated in order to explain how a supervised neural network works:

Imagine teaching a little child what a ball is. Firstly, a ball is showed to him, explaining to him that it is a ball. Teaching him different kinds of balls, the child figures out that it is the shape of the ball that makes it a ball and not the colour, size or texture. Then, if an apple is displayed and the child is asked about it, answering that it is a ball, the teacher will say no, identifying it as an apple. Repeating this process, he will be able to distinguish between a ball and an apple. Finally, if a ball is given to him, he will classify it properly.

A Supervised algorithm works in the same way. Several thousand samples of the classes are shown with their labels, identifying the object. Therefore, to implement the algorithm, a set of data with a wide range of images and their corresponding names is required. The more data there is, the better the model will be able to learn and thus, function properly. After this information is recorded, the process can begin.

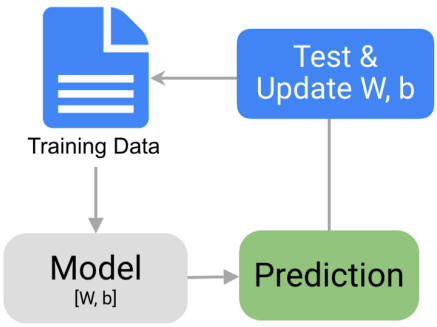
Initially, the set of data is split into training sets, one of which will be of the majority data for the model, and the other one, will be the test set. The reason for this split is that the networks are in danger of memorizing their inputs during the training. By keeping the test set separate, we can ensure that the model works with data that it has never seen before, (Google, 2018). Giving an example, where for each input x there is an output y:

$$x \longrightarrow y \tag{2.1}$$

The predictor tries to create a model which gives the same values, trying to be as exact as possible.

$$h(x) = f(x) \tag{2.2}$$

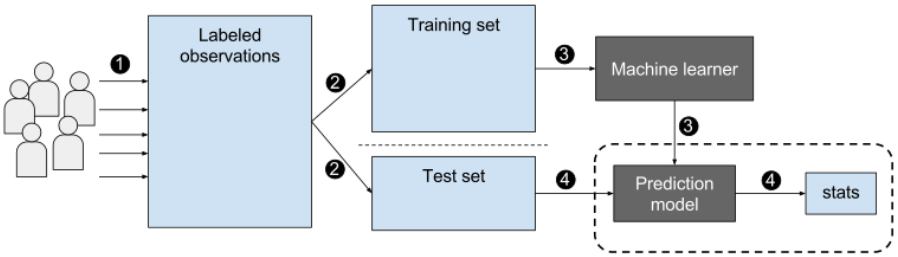
In the training step of a model, there are input training data *x_train* with their corresponding output y. For each sample the discrepancy between the prediction *h(x_train)* and the real output y is computed. This difference is a measure of the prediction accuracy and, with enough training data, the margin of error will decrease progressively. The difference is named Cost Function and there are many different functions that are able to calculate the error. However, the goal is always the same and the process is repeated until the model has converged with the best values.



Source:
https://towardsdatascience.com/the-7-steps-of-machine-learning-2877d7e5548e

Figure 2.4 Training step in a Supervised Machine Learning task.

After a predictor, the prediction model has been processed by using the training set and the model can be proofed by making predictions against the test set. Because the data in the testing set already contains known values for the attributes that you want to predict, it is easy to determine whether the model's guesses are correct or not, (Mircrosoft SQL Server, 2012). After these models are created, they have the ability to recognize patterns in the data base and this is called pattern recognition.



Source:
https://commons.wikimedia.org/wiki/File:Supervised_machine_learning_in_a_nutshell.svg

Figure 2.5 Supervised Machine Learning structure.

Furthermore, some applications use an additional third set of data, which is called the validation set. This set is a process that take place in between the training set and the test set and its goal is to provide a better performance of the model. This set will be employed in the algorithms implemented in Section 4.2. In our neural networks, the inputs are the relative positions and velocities of the UAVs at different simulation steps, the distance to the obstacles and the distance to the goal position for the UAV that wants to be guided. The output is the next velocity given to the drone in order to reach the goal position and avoid collisions with other drones or obstacles.

Since its origin, the NNs have become an important tool and nowadays, there are thousands of applications that use these algorithms to answer their tasks. Facebook uses it to automatically find and tag friends in photos, Skype utilizes it to transmit spoken conversations in real time and even Google Translator applies it to translate sentences from a language to another.

Basic parameters

Although, a NN can be really complex, the basic variables that are implemented in this paper are going to be briefly explained. This will help us to find more optimal architectures.

- Number of instants: number of movements made by a drone in the different simulations. It can be also explained as the number of rows in the input matrix of the algorithm. If the size is too big, the NN could be overfitted, an idea that will be later explained.
- Training, validation and test size: lengths of the input data set that will be used to train, validate and test the model respectively. If the training is too small, the NN will have not enough data to train.
- Batch size: number of training examples present in a single batch. This value depends on the training size. If the training is big enough, the batch size can be also bigger.
- One Epoch: when a complete data set is passed forward and backward through the NN only once. The number of epochs is directly proportional with the response time. It will be necessary to find the optimal number of epochs.
- Iterations: the number of batches needed to complete one epoch.
- Learning rate: this parameter informs the optimizer how far to move the weights in the direction of the gradient for a small batch. If it is too big, the algorithm will be never precise, causing the training to not converge. However, if it is to small the optimization will take a lot of time, because steps towards the minimum of the function loss are tiny.
- Standard deviation: it is a measure used to quantify the amount of variation or dispersion of a set of data values. It will appear in the initialization of the weights and it is set by default to 1. A low standard deviation indicates that the data points tend to the expected value. While a high value, means the opposite.
- Training algorithm: it is the procedure used to carry out the learning process in a neural network.
- Activation function: function that introduces non-linearity into the output of each neuron. Otherwise, the model will end up linear, which could cost a failure in the classification task. The most popular

activation functions are Rectifier Unit (ReLu), which should be only applied in hidden layers, Sigmoid Activation function (Sigmoid) and Hyperbolic Tangent function (Tanh).

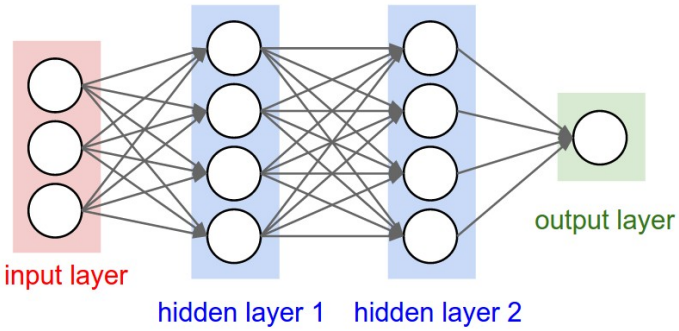
Regarding the running time of a model, it depends on the time that the optimization process takes to "teach" the NN. This is conditioned by the parameters of the model and also the hardware and software used for the "learning". In our case, a MSI GL62M computer with an Intel(R) processor Core(TM) i7-7700HQ CPU 2.80 GHz, a RAM memory of 8,00 GB and a Solid state drive (SSD) of 256 GB was employed. The software used is Ubuntu 16.04 LTS, installed in the SSD, and TensorFlow with Python3 on it. This information is important in order to know how fast the computer works for the different neural networks implemented. Therefore, it is a significant variable to consider when a model is being optimising and also when it is going to be simulated.

The model depends on the settings of these parameters as this will be shown in Section 4.2.

Architectures

There are several NN architectures that can be applied for different tasks. In this paper, a Fully Connected architecture will give sufficient results according to the results in Section 4.2 and 5. Despite this, some of the most common architectures are also commented on.

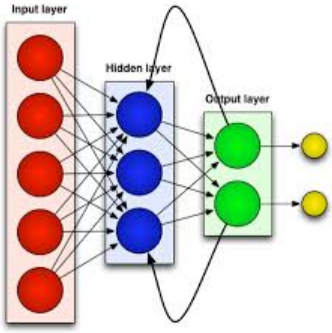
- Fully Connected layers is an architecture composed of different hidden layers between the input and the output.



Source:
http://cs231n.github.io/convolutional-networks/

Figure 2.6 Fully Connected neural network.

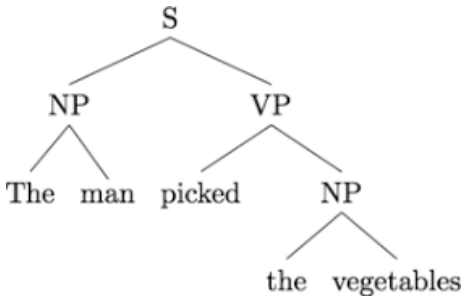
- Recurrent Neural Network is a standard neural network extended across time by having edges which feed into the next time step. This mean that RNN uses sequential information, performing the new output according to the information that has been calculated so far. For example, RNN is implemented to predict the next word in a sentence, based on the word before, (*Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs – WildML*, 2000). This method is really useful for machine translation and speech recognition. This architecture is illustrated in Figure 2.7.



Source:
https://jasonroell.com/2017/06/26/understanding-recurrent-neural-networks-the-prefered-neural-network-for-time-series-data/

Figure 2.7 Recurrent Neural Network architecture.

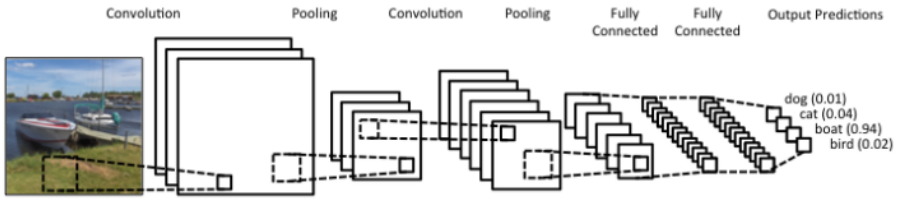
- Recursive Neural Network is structured like a hierarchical network, with no time aspect, such as in Recurrent Neural Networks. Here, the input has to be processed hierarchically in a tree fashion. An example of how it looks like is pictured in Figure 2.8.



Source:
https://nlp.stanford.edu/blog/hybrid-tree-sequence-neural-networks-with-spinn/

Figure 2.8 Recursive Neuronal Network example.

- Convolutional Neural Network (CNN) is based on a feedforward artificial neural network system, which means that the information moves only forward in the graph. It has several layers which are grouped into five different blocks: Input layer, Convolutional layers, Rectified Linear Unit as an activation function, Pooling layers and Fully Connected layers. The Figure bellow 2.9 shows the architecture commented on previously.

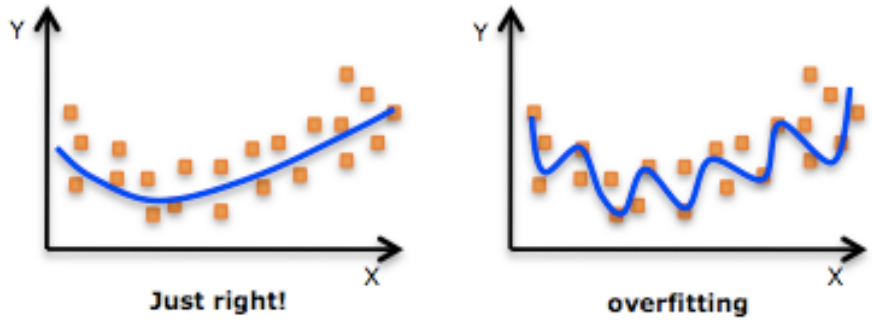


Source:
https://www.kdnuggets.com/2017/11/understanding-deep-convolutional-neural-networks-tensorflow-keras.html

Figure 2.9 Convolutional Neural Network architecture.

Overfitting

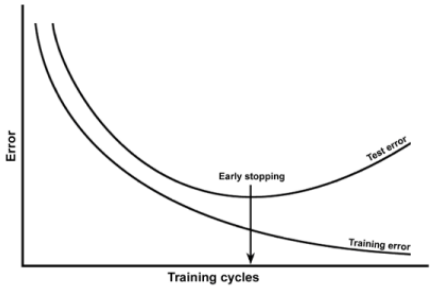
Overfitting is a neural network error that occurs when a model fits in too much seen data and does not generalize well. This can be observed when the performance on test sets is much lower than the performance on training sets. Another sign of overfitting is that the test error, at one point, increases while the training error continuously decreases. In Figure 2.10, we can observe the difference between a model which is learning correctly and another that is overfitted.



Source:
https://www.quora.com/Whats-the-difference-between-overfitting-and-underfitting

Figure 2.10 Overfitting.

Some of the techniques employed to avoid overfitting in a neural network are, for example, Early Stopping or Regularization. The first one refers to stopping the training process before the model passes the point in which the model's ability to generalize can weaken as it begins to overfit the training data. An example of this, can be seen in Figure 2.11. The second one, Regularization, adds a penalty to the different parameters of the model to reduce the freedom of it. This model will be less likely to fit the noise of the training data and will improve the generalization abilities of the model. In this paper and depending on the model into study, it will be used regularization as technique to avoid overfitting.



Source:
https://chatbotslife.com/regularization-in-deep-learning-f649a45d6e0

Figure 2.11 Early Stopping technique.

Tensorflow

TensorFlow is faster, smarter, and more flexible than our old system (DistBelief), so it can be adapted much more easily to new products and research.

GOOGLE

Introduction

TensorFlow is an open-source software library for numerical computation using data flow graphs. It was developed by Google Brain after DistBelief in November 2015. DistBelief comes from Large Scale Distributed Deep Networks and was the first machine learning system based on Deep Learning Neural Networks built by Google Brain in 2011. This framework was simplified into a faster and more robust application-grade library, which now is TensorFlow.

The TensorFlow library was created for the purposes of conducting machine learning and deep neural networks research by Google Brain, but nowadays it is also useful to compute a highly complex number of operations, (Dean et al., 2000).

Figure 3.1 Logo Tensorflow.

The name TensorFlow comes from tensor, which is a generalization of vectors and matrices to compute potentially higher dimensions. It is defined with a shape, which consists of the number of elements in each dimensions of the tensor and a data type. Accordingly, each element in it has the same properties, (*Tensors | TensorFlow*, 2000).

TensorFlow corrects the shortcomings of DistBelief by making a general, flexible, portable, easy-to-use, fast and completely open source.

TensorFlow can run on one or more CPUs or GPUs in a desktop, server and mobile computing platforms like Android and iOS, due to the flexible architecture on which it was built. Its portability enables researchers to move an idea from training in a GPU or CPU to running on a mobile phone. Moreover, TensorFlow has APIs (Application Programming Interface) available in several languages like Java, C++, which has a low overhead, or Python. Currently, Python API is the most complete and the easiest to use, (Tensorflow, 2017).

Tensorflow is not only a deep learning library, it is also a number-crunching framework. The different between TensorFlow and other libraries like Theano, is that TensorFlow allows us to perform specific machine learning number-crunching operations like derivation on huge matrices with large efficiency. For this reason, although TensorFlow is mainly used with machine learning right now, it could be used in other areas, since it is really a massive array manipulation library, (*Python Programming Tutorials*, 2000). TensorFlow can be used for any computation that could be expressed as a computational flow graph.

There are other frameworks that can be used to build deep learning solutions, some of which are quite complex, such as Keras or PyTorch. The first one can also be configured to work on top of TensorFlow, which is written in Python and is easy and straightforward to learn. PyTorch, on the other hand, is a Python package for building deep neural networks and performing complex tensor computations. Its modelling process is easy and transparent. However, TensorFlow has some advantages better than other frameworks, giving reasons to why this is implemented instead of others.

Structure

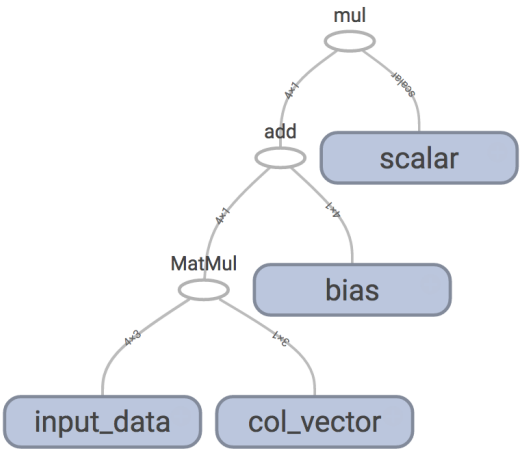
TensorFlow consists on a number of ordered operations using a dataflow graph to represent the dependencies between individual operations. In general, a dataflow graph is defined with nodes, where each node has input/output data ports and edges which are the connections between these ports. In TF, the nodes represent various operations, including mathematical functions such as addition and multiplication, variable operations for storing model parameters and initialising the tensor values. The graph edges have three different aims. The data dependency edges which are represented as tensors or multidimensional arrays, that are input and output data of the operations. It is important to know that tensors don't have value and they are just handles to elements in the computation graph. Another type of edge are the reference edges, which represent pointers to the variables instead of its value. The third type are the control dependency edges that indicate their source operations must execute before their tail operations can start, (Wongsuphasawat et al., 2000).

These graphs have multiple pros that TensorFlow utilizes when executing the programmes.

- Parallelism: explicit edges represent the dependency between different operations, making the identification of the system easier regarding which operations can be executed in parallel or not.
- Distributed execution: this property helps TensorFlow to run on multiple devices, inserting the necessary communication and coordination between them.
- Compilation: TensorFlow compiler generates faster codes by using the information in the dataflow graph, fusing them together, for example, adjacent operations.
- Portability: The representation does not depend on the programming language used.

These advantages allow TensorFlow an easy computation and analysis of the models.

An illustration of a TensorFlow graph can be as follow in Figure 3.2:



Source:
https://clindatsci.com/blog/2017/5/31/distributed-tensorflow

Figure 3.2 Dataflow graph in TensorFlow using TensorBoard visualization tool.

In addition, TensorFlow provides the class Session to represent a connection between the client program, typically a Python program, and the C++ runtime. A session evaluates tensors while it encapsulates the state of the TensorFlow runtime and runs TensorFlow operations. This supplies access to devices in the local machine and remote devices using the distributions of TensorFlow runtime, (Google LLC, 2018). Moreover, sessions are useful since their cache information about the graph, so that it is possible to efficiently run the same computation several times. Therefore, a session gives us the possibility of training a neural network only once and then be able to save, import and restore the graphs simply by adding a few code lines. There are two different files to store the data of the graph, depending on what we want to save, Checkpoint file and Meta graph. The first one is a binary file which contains all the variables saved, like gradients or weights, while Meta graph saves the complete TensorFlow graphs. The second one mentioned will be the one used in our models. We will create the models and then, after they are optimised, import and restore them to test their performance in the Gazebo simulator which will be described in Section 5.

Visualization tool: TensorBoard

Sometimes a neural network could be defined as a black block, and here is where TensorBoard appears. It is a TensorFlow tool to visualize dataflow graphs, show additional data that pass through it and plot quantitative metrics about the execution of the graph. It can be considered as a "flash light" into this black block, giving the possibility to dive in the program.

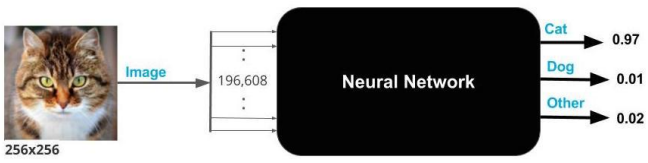


Figure 3.3 Neural network like a black box.

In Figure 4.1, the image of a cat is the input and after the neural network box, the output with the prediction is read. Tensorboard would be like a magnifying glass of the box.

TensorBoard operates by reading TensorFlow events files, which contain summary data which can be generated when running TensorFlow (TensorFlow, 2017). Depending on how the data lifecycle of the model wants to be showed, a different type of summary will be used. It supports five visualizations: scalars, audio, histograms, images, and graphs. For example, to record how the learning rate varies over time and how the objective function is changing *tf.summary.scalar* could be useful. If it is wanted to visualize the distributions of gradients or weights in the model, *tf.summary.histogram* is a good option. Normally the computations can be complex and difficult to understand but with TensorBoard's help, it is easier and more intuitive.

To make the visualization even more simple and organized, variable names can be scoped and then this information is used to define a hierarchy between nodes in the graph. Only the top of this hierarchy is shown by default. To scope the variable, $tf.name\_scope()$ is used, encapsulating all variables that are defined inside. The next Code is an example of addition using the scope named *block*.

Code 3.1 Addition example with TensorBoard.

```
with tf.name_scope('block') as scope:
  a = tf.placeholder(tf.int8, name="a")
  b = tf.placeholder(tf.int8, name="b")
  addition = tf.add(a, b, name="addition")
sess.run(addition, feed_dict={a: 2.0, b:3.0})
```

After that, it is possible to visualize it in a TensorFlow graph with TensorBoard by adding a few more code lines. This could help if the results of the model are not the appropriated ones to know where the error is.

A class, whose name is FileWriter, allows to write data from TensorFlow to disk so that it is possible to read it. In other words, it enables to write any information that is going to be shown in TensorBoard. The next lines are written to create the TensorFlow graph where the collected summary data are going to be taken. The constructor in $tf.summary.FileWriter()$ is the direction where all of the events are going to be saved:

```
writer = tf.summary.FileWriter("path/to/log-directory")
```

When the training has already finished, we need to write the next line in the Terminal of the computer, in order to call TensorBoard with the path to the directory where the data are saved:

```
tensorboard --logdir /path/to/log-directory
```

To finish, and without closing the Terminal, we need to search for it in a web browser:

```
localhost:6006
```

What comes now in the browser is TensorBoard with the different visualization types that were computed, in this case, only the Graph part:

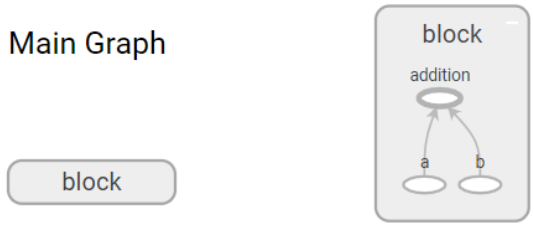


Figure 3.4 Example of name scopings and nodes.

To see how TensorBoard can help the computations, the neural networks trained in Section 4.2 are going to be illustrated in the browser with this tool.

TensorFlow nowadays

Currently, TensorFlow is the number one repository in GitHub with around 5500 GitHub repositories with "TensorFlow" in the title and has about 475 non-Google contributors. This indicates how well TensorFlow has been received since it was created. Now, there are also universities which start using TensorFlow as the basis of important machine learning classes.

TensorFlow can be applied in different areas as a powerful calculation framework, but in this paper it is essentially used in Deep Learning algorithms.

TensorFlow is mainly used for: classification, perception, understanding, discovering, prediction and creation. There are some different applications such as:

- voice/sound recognition such as Apple's Siri or Google Now for Android;
- text based applications to detect languages like in Google Translate;
- image recognition targets to recognize and identify people and objects in images as well as understanding the content and context;
- TensorFlow Time Series algorithm are used for analysing time series data in order to extract meaningful statistics;
- video detection like in Motion detection, Real-Time Thread Detection in Gaming and security, (*Top Five Use Cases of TensorFlow*, 2000).

There are a lot of common applications using Machine Learning and TensorFlow, like spam detection in Gmail and signal understanding in Street View. Companies such as Airbnb, Airbus, Twitter or Dropbox are using TensorFlow to solve their own necessities.

Training of the scenarios

More data beats clever algorithms, but better data beats more data.

PETER NORVIG

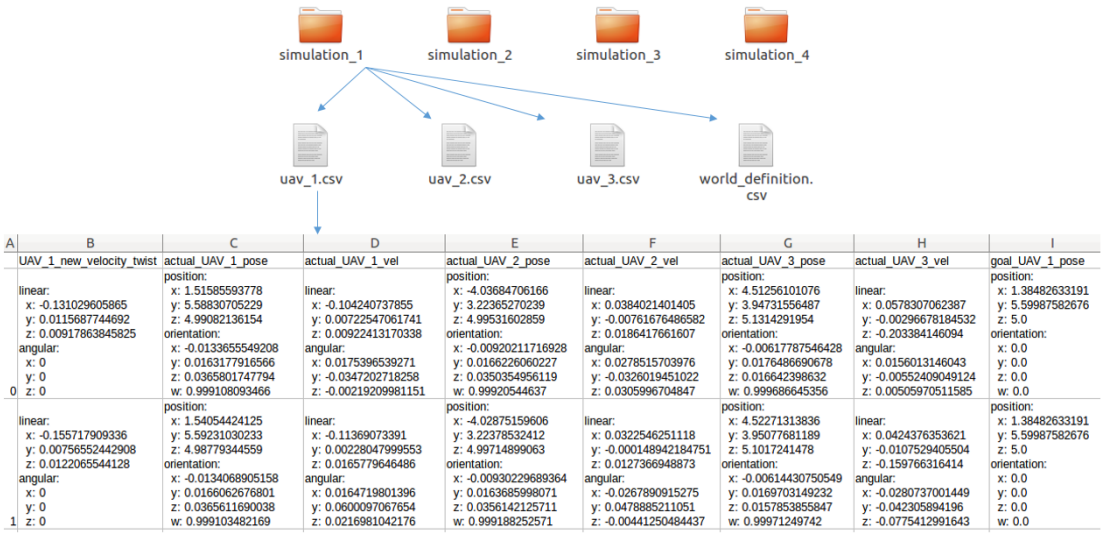
Dataset

In order for neural networks to work, the data need to be trained. This information has been created by a simulator provided and implemented by a member of the investigation group of the department. With the help of UAV Abstraction Layer of GRVC, the student designed his own simulator with Gazebo and ROS. UAL is a tool implemented in ROS to interact with the autonomous pilot of the UAV and Gazebo in an easy and intuitive way. This simulator computes and simulates scenarios with information from the algorithm ORCA, an algortihm used to avoid obstacles. This algorithm is also from another project done at the University of Seville and is not part of this study here. The results of the simulations made with ORCA are the dataset used in this research in order to be able to train the neural networks. As commented on, in the Section 1.2, the objective of this paper is to examinate if we can use neural networks created with TF to guide drones in an easy and efficient way. Therefore after the algorithms are computed, as we can see in Section 5, this simulator is again employed to check the performance of the UAV in the world.

Once we are informed of the origin of the dataset, it is necessary to learn its structure, organization and extraction process in order to know how it is employed. Each ORCA simulation is done with a specific scenario and when the simulation has finished, this information is stored and saved in a file called *world_definition.csv*. The information needed to be extracted from this file are the number of obstacles with their coordinates positions and the number of UAVs. This knowledge is essential to get the size of the neural network that is implemented. The size corresponds to the number of input and output features of the algorithm. There will be as many NNs trained as there are different scenarios to be simulated with Gazebo.

At the same time for each simulation, files are created with the positions and velocities of the different drones, the goal position, and the next velocity applied to the reference UAV. The name of these files follows the structure *uav_x.csv* where *x* in the number of the UAV that is trained according to the position and velocity of the others. For example, a simulation with three UAVs will have three files, one for each of the UAVs in the simulation. Thereafter, the reference UAV will be also called UAV_1.

The dataset of each *y* simulation is contained in different folders named *Simulation_y*. Inside these folders, the file *world_definition.csv* and the file or files *uav_x.csv* are stored. The next Figure illustrate how these files are stored for an easier understanding.



A	B	C	D	E	F	G	H	I
	UAV_1_new_velocity_twist	actual_UAV_1_pose	actual_UAV_1_vel	actual_UAV_2_pose	actual_UAV_2_vel	actual_UAV_3_pose	actual_UAV_3_vel	goal_UAV_1_pose
0	linear: x: -0.131029605865 y: 0.0115687744692 z: 0.00917863845825 angular: x: 0 y: 0 z: 0	position: x: 1.51585593778 y: 5.58830705229 z: 4.99082136154 orientation: x: -0.0133655549208 y: 0.0163177916566 z: 0.0365801747794 w: 0.999108093466	linear: x: -0.104240737855 y: 0.00722547061741 z: 0.00922413170338 angular: x: 0.0175396539271 y: -0.0347202718258 z: -0.00219209981151	position: x: -4.03684706166 y: 3.22365270239 z: 4.99531602859 orientation: x: -0.00920211716928 y: 0.0166226060227 z: 0.0350354956119 w: 0.99920544637	linear: x: 0.0384021401405 y: -0.00761676486582 z: 0.0186417661607 angular: x: 0.0278515703976 y: -0.0326019451022 z: 0.0305996704847	position: x: 4.51256101076 y: 3.94731556487 z: 5.1314291954 orientation: x: -0.00617787546428 y: 0.0176486690678 z: 0.016642398632 w: 0.999686645356	linear: x: 0.0578307062387 y: -0.00296678184532 z: -0.203384146094 angular: x: 0.0156013146043 y: -0.00552409049124 z: 0.00505970511585	position: x: 1.38482633191 y: 5.59987582676 z: 5.0 orientation: x: 0.0 y: 0.0 z: 0.0 w: 0.0
1	linear: x: -0.155717909336 y: 0.00756552442908 z: 0.0122065544128 angular: x: 0 y: 0 z: 0	position: x: 1.54054424125 y: 5.59231030233 z: 4.98779344559 orientation: x: -0.0134068905158 y: 0.0166062676801 z: 0.0365611690038 w: 0.999103482169	linear: x: -0.11369073391 y: 0.00228047999553 z: 0.0165779646486 angular: x: 0.0164719801396 y: 0.0600097067654 z: 0.0216981042176	position: x: -4.02875159606 y: 3.22378532412 z: 4.99714899063 orientation: x: -0.00930229689364 y: 0.0163685998071 z: 0.0356142125711 w: 0.999188252571	linear: x: 0.0322546251118 y: -0.000148942184751 z: 0.0127366948873 angular: x: -0.0267890915275 y: 0.0478885211051 z: -0.00441250484437	position: x: 4.52271313836 y: 3.95077681189 z: 5.1017241478 orientation: x: -0.00614430750549 y: 0.0169703149232 z: 0.0157853855847 w: 0.99971249742	linear: x: 0.0424376533621 y: -0.0107529405504 z: -0.159766316414 angular: x: -0.0280737001449 y: -0.042305894196 z: -0.0775412991643	position: x: 1.38482633191 y: 5.59987582676 z: 5.0 orientation: x: 0.0 y: 0.0 z: 0.0 w: 0.0

Figure 4.1 Data set files for simulations with 3 UAVs.

The data set of a specific NN will be all the information stored in simulations of the same type of world. Therefore a huge data set is created by using different situations for the same scenario. Although, the positions and velocities are stored in 3D values, the inputs and outputs are reduced to 2D, x and y values, in order to make the problem less complex. The code where the information is extracted and then structured to compute the inputs and outputs of the algorithm is in Code 7.1 and 7.2, respectively.

It is important to note, that every simulation employed to "teach" a neural network can not have any errors, which means without any collisions. If the algorithm understands as correct a simulation where there are collisions, it will not learn what we wanted it to and the drone could drive directly into an obstacle. Therefore, before using the data provided by ORCA, we have to check the file *performance_set.csv* where each simulation is associated with a true or false variable, depending on its correct or incorrect performance. Here, correct means without collisions and incorrect the contrary. The reason for this process of filtration will be demonstrated in Section 5.4 where we will show the different results between using this filter and not. The filter is implemented in Code 7.3. These errors in the data set come from the algorithm ORCA, which is not part of this study.

Once the data set is understood and filtrated, the next step is to shuffle the information, in order to make the learning more efficient. Then, it is divided in a training, validation and test set. Normally, this separation follows the rule of 70% - 15% - 15% for training, validation and test set, respectively. This will be the division employed in every algorithm used.

In general, the inputs of the NNs are the relative position of each UAV, their velocities and the distance to the goal position and to each obstacle for the reference UAV. The outputs are the new velocities applied to the UAV 1, also called UAV of reference. Depending on the scenario where the NN is employed, it will have more or fewer inputs, while the outputs will always be of the same size, in this case, 2.

Training

In this Section, we will compute algorithms for the different scenarios, trying to find the most optimal network for each one. In order to trust the results, all the neural networks implemented are checked at least five times with the same configuration to avoid random results and verify the algorithm. An accuracy of 100% means that there is no error in the test set and the drone will move to the goal position without collisions. While an accuracy smaller than 60% means that the NN is not learning at all.

In all the models, Backpropagation algorithm is employed to better the performances. Backpropagation means that after the error between predictions and outputs is calculated, this information goes back into the network. Depending on the contribution of each neuron to the output, each one will take a part of the total error modifying their values. This algorithm helps training algorithms, like Gradient Descent, to obtain a lower error along the process. In addition, the collision filter will always be employed except in Section 5.4 where we will inspect to see if the filter is actually relevant or not.

To start with, a simple neural network for a scenario with one drone and no static obstacles is implemented. This will give us a good understanding of how the different parameters change the results. These variables were explained in Section 2.3. After this model, it will be easier to implement more complex neural networks.

First neural network. 1 UAV and 0 obstacle

This algorithm will have four inputs in 2D, the distance to the goal position and the velocity of the UAV; while the output is the velocity in x and y axis. Here, the data set has already passed the filter commented on before and therefore the network will only "learn" correct data. This information is illustrated in Figure 4.2.

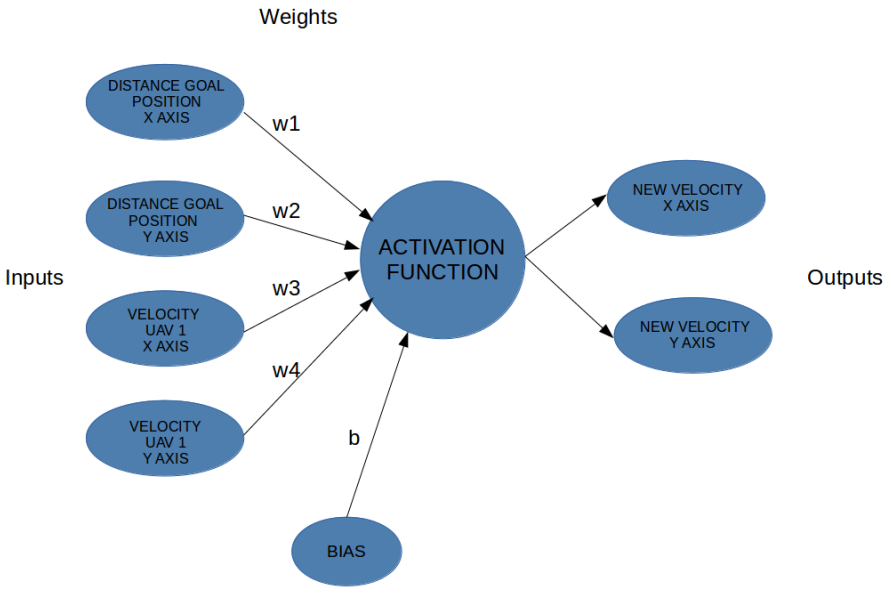


Figure 4.2 Architecture of the neural network (1 UAV and no obstacles).

At first, the main idea was to find the lowest number of instances needed to compute the algorithm with the best performance. After several tests, the lowest number was approximately 800 instances. Working around this configuration and trying to find the best parameters, we end up with the model shown in Figure 4.2. The batch size and the number of epoch sets was 3 and 200, respectively. The training algorithm that was used was a Gradient Descent with a learning rate of 0.001. Also, the standard deviation (stddev) was fixed to 0.3 and the activation function implemented was a hyperbolic tangent. This model was able to finish its optimization process in 11 seconds, with an error between 0 and 2% . These values are considered optimal and, therefore, the basis of our study.

After this model was computed, we end up with some ideas about how some parameters affect the accuracy of the model.

- A small number of instances will cause that the model to not learn enough while a large number may cause overfitting.
- After the optimal point, a bigger number of epochs will only cause that the optimization process to need more time.
- A high stddev value causes a lot of dispersion in the distribution of the weights or, in other words, poor values of the initial weights causing a unsuccessful training.
- A low learning rate will cause a longer optimization process time. This happens because steps toward the minimum of the loss functions are tiny. On the contrary, a value that is too large will cause the training to not converge or even to diverge.
- If the data set is big enough, we can use a bigger batch size in order to avoid generalization.

Now, the algorithm is modified to illustrate its structure using TensorBoard. This will give us a better understanding of how it works and how the different parameters are interconnected. The next Figure 4.3 shows its main graph composed only of an activation function. This layer is fed into the weights and the bias, returning a function loss between the prediction and the known output.

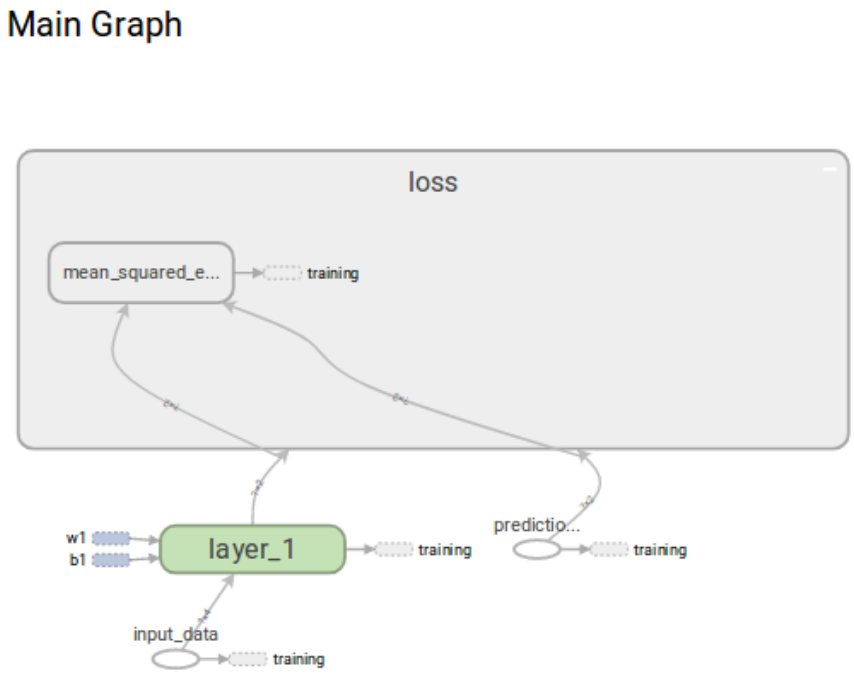


Figure 4.3 Tensorboard data flow main graph (1 UAV and no obstacles).

In addition, in Figure 4.4, we can see its auxiliary graph where the validation and test set is predicted. Inside every node, there are different operations and connections. This is why TensorBoard is considered a "flashlight" in the algorithm and a very interesting tool.

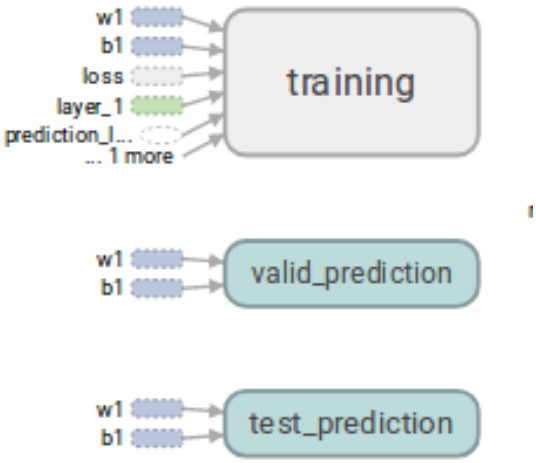


Figure 4.4 Tensorboard data flow auxiliary graph (1 UAV and no obstacles).

It is also possible to see how scalar parameters change along the optimization process, like for example the accuracy and the total loss of the model. While the function loss should decrease, trying to converge to zero, the percent accuracy gets better the bigger the values are. Below, it is shown how the accuracy is converging to 100%.

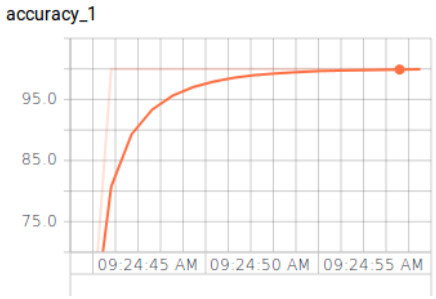


Figure 4.5 Accuracy (1 UAV and no obstacles).

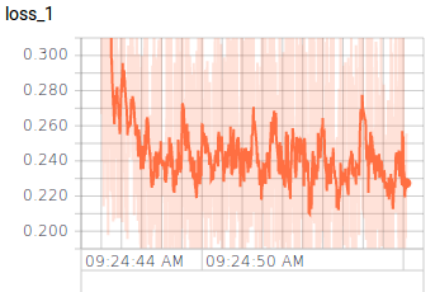


Figure 4.6 Function loss (1 UAV and no obstacles).

Later, in Section 5, we will run this algorithm in order to see how this NN guides the drone in the simulator. Although there is no obstacle to collide with in this scenario, it is a good starting point in the research of more difficult and complex algorithms. After this neural network model is looked into, we are able to go one step forward, implementing a model with one UAV and one obstacle.

1 UAV and 1 obstacle

In this case, there are six inputs in the NN due to the position of the obstacle. The idea was to start with the previous model and gradually increase its depth.

First, choosing a good data set length, is necessary for the base of the model that is going to be optimised. We try to use as much data as the model can handle and learn without becoming overfitted. Several training sessions were done and the length of the data set employed was changed each time. At the end, the number of moments of the data set was set up to around 16,000, which were stored in 118 different simulations. This size was chosen because the corresponding accuracy of the model was good enough to simulate with a small error. However, it will also work with fewer steps. This will be the starting point of the optimization process.

After that, the next step is to define the rest of the parameters following the configurations with better results. Nevertheless, we soon realised that the previous architecture is, for this case, too simple. Therefore and due to the complexity of this task, it is necessary to go "deeper" into the architecture, creating hidden layers. This type of model corresponds to a Fully Connected layers architecture which was introduced in Section 2.4.

The model that it was chosen to be implemented as the best one has three hidden layers with 30 neurons in each one. Nevertheless, more layers mean more depth and therefore more precision; it also makes the optimization process longer and at a higher computation cost. The upcoming Figure 4.7 shows the optimal architecture selected:

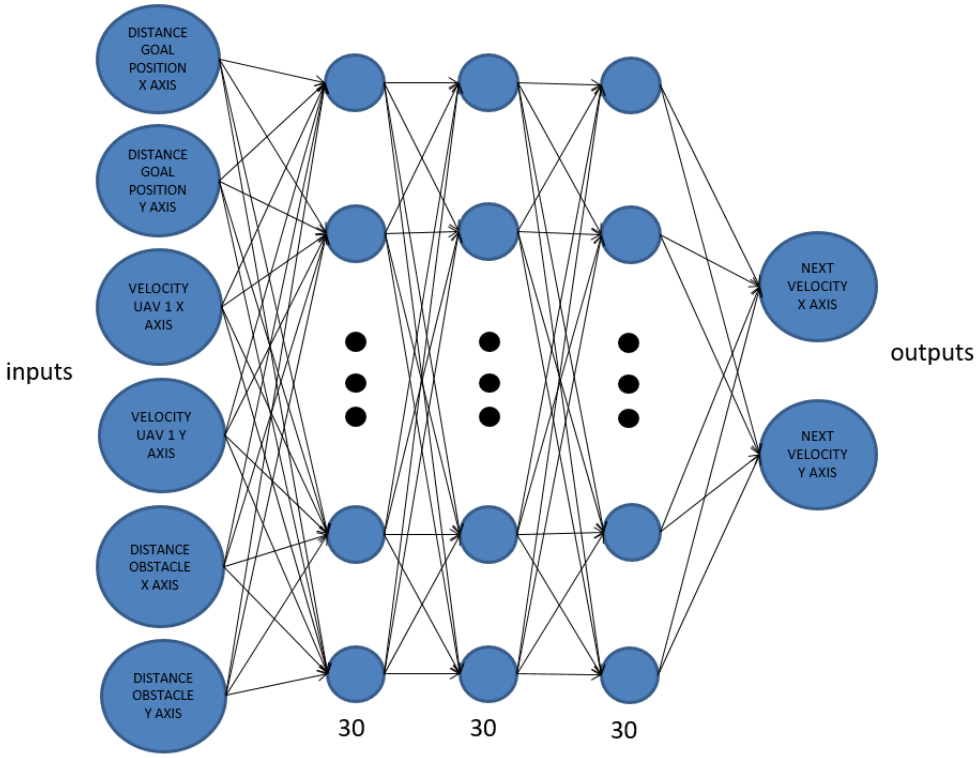


Figure 4.7 Architecture of three Fully Connected layers(1 UAV and 1 obstacle).

Once the architecture has been chosen, we can set up the remaining parameters in order to have the best performance. The optimal configuration that was found is the following:

- data set of 16,000 instants;
- three hidden layers with 30 neurons each one;
- number of epochs 800 and batch size 20;
- learning rate of 0.006;

- standard deviation for the first layer of 0.7. The sttdev for the rest of layers is normally set with the equation $\sqrt{\frac{2}{num\_neurons}}$;
- Tanh Activation functions for every layer. The rest of functions decrease the accuracy;
- Gradient Descent algorithm.

This configuration gives us an accuracy around 97%, which could be sufficient for the simulation. This range was determined by carrying out ten different simulations. The neural network implemented for this scenario is computed in Code 7.4.

The next Figure 4.8 illustrates its dataflow main nodes with the different hidden layers and the function loss. Now, the architecture is deeper and, therefore, its graph is more complex than in the previous scenario.

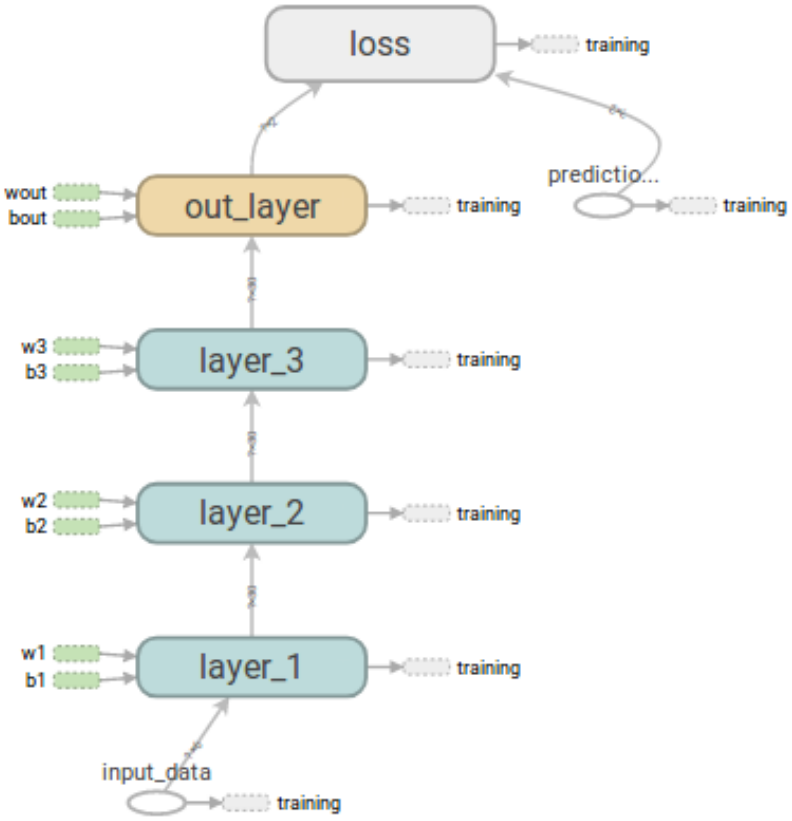


Figure 4.8 Tensorboard data flow main graph (1 UAV and 1 obstacle).

Now, in the auxiliary nodes of Figure 4.9, we can see how the Gradient Descent is implemented and how the validation and test set is predicted for this scenario.

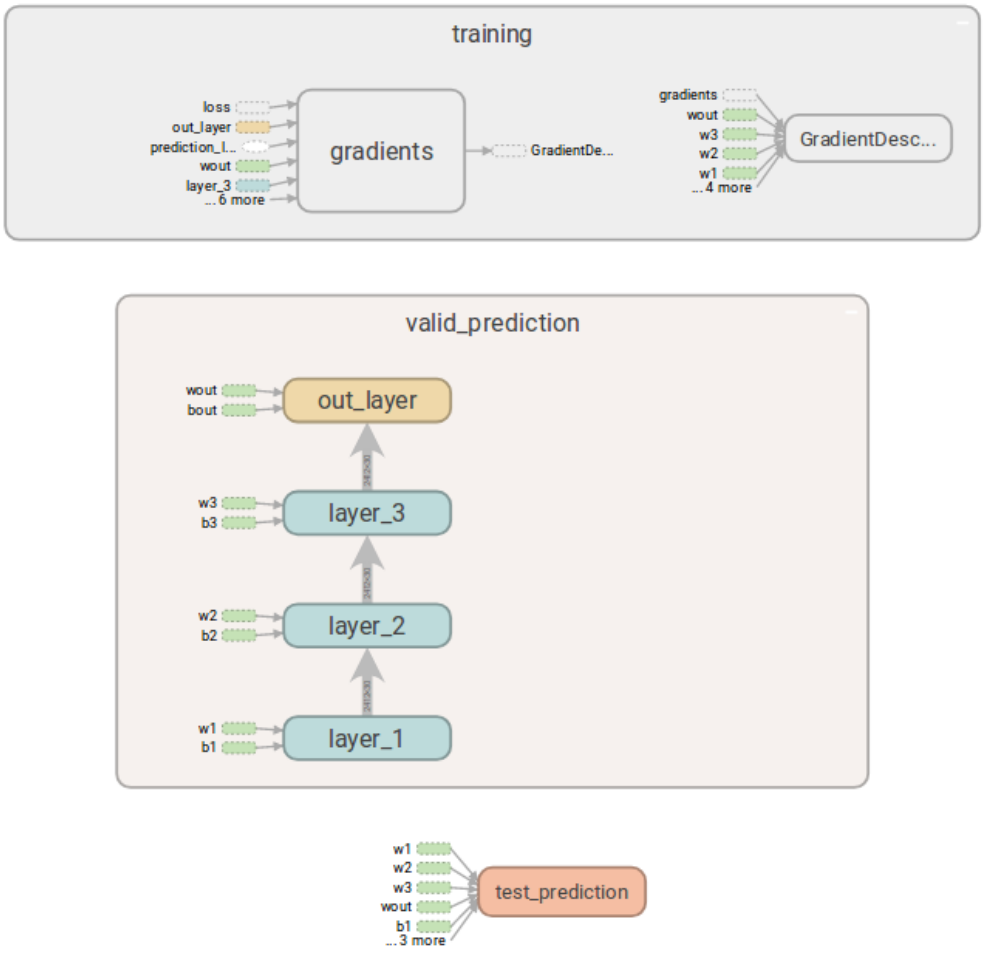


Figure 4.9 Tensorboard data flow auxiliary graph (1 UAV and 1 obstacle).

In this case, the accuracy and function loss graphs are as follows, converging the accuracy to 97% as was previously mentioned. The optimization time is around 3 minutes, but the final accuracy is achieved in approximately 2 minutes.

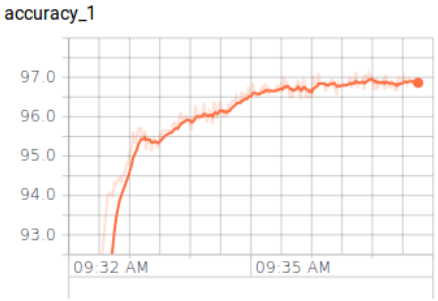


Figure 4.10 Accuracy (1 UAV and 1 obstacle).

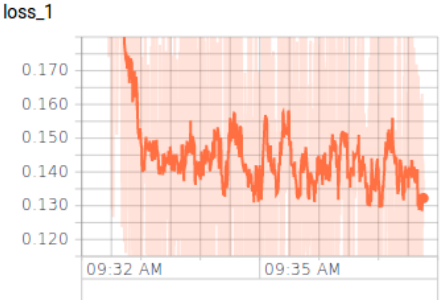


Figure 4.11 Total function loss (1 UAV and 1 obstacle).

2 UAV and 0 obstacle

Although, the 2 UAVs and the obstacle both contribute to the input matrix with their relative positions, the UAVs also affect the matrix with their velocities. Therefore, in this scenario we will have 8 inputs for the network trained.

Here, the model achieved, which gives a relatively good learning percent, is an algorithm trained with around 5,600 instants and with an architecture of three hidden layers (40 neurons in each one)as it can be seen in Figure 4.12.

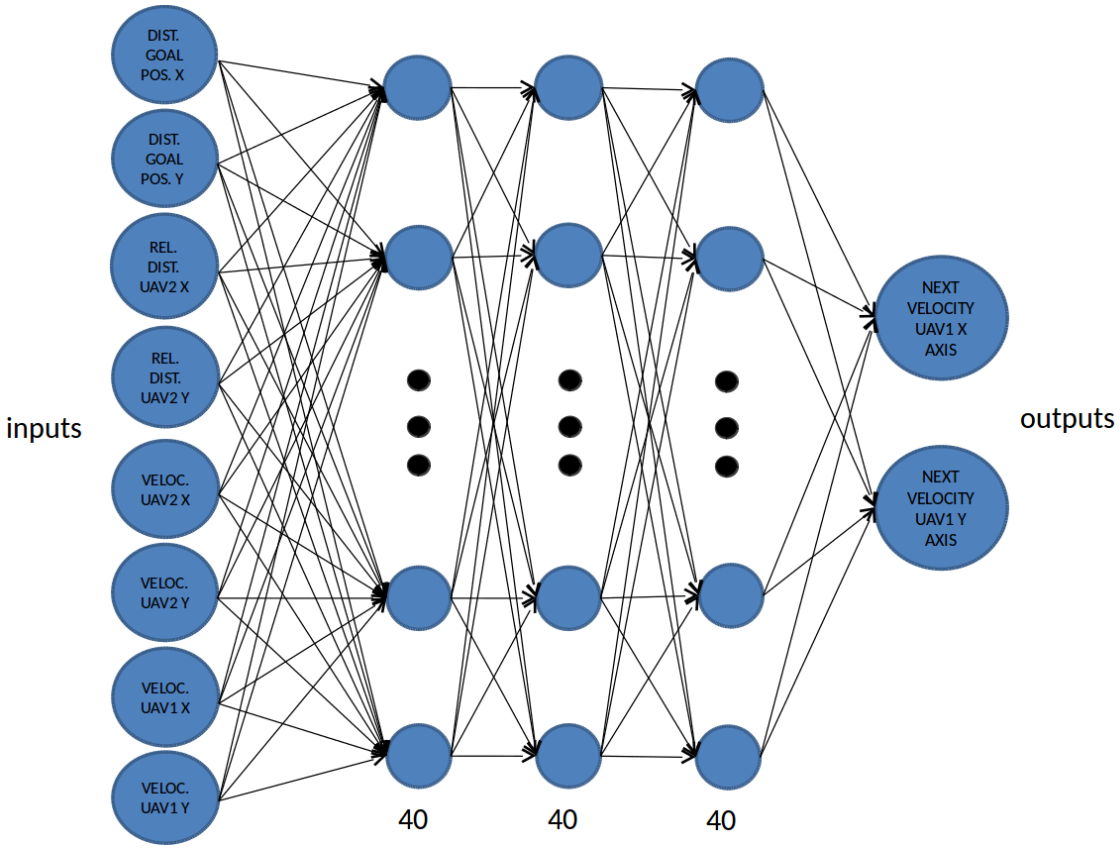


Figure 4.12 Architecture of three Fully Connected layers (2 UAVs and no obstacles).

Furthermore, we have also changed the number of epochs to 1,200 and the batch size to 40 in order to avoid overfitting. Nevertheless this problem is not eliminated with this model and, therefore, it is necessary to compute a technique to specifically avoid it. As previously mentioned, we have employed Regularization, which normally gives good results. After that, the learning accuracy approximates 97%. This can be demonstrated by illustrating the accuracy of the model with TensorBoard.

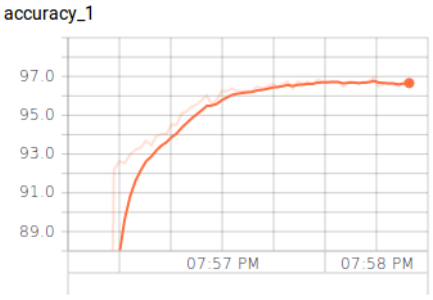


Figure 4.13 Accuracy (2 UAVs and no obstacles).

In the next figures, the function loss of this neural network is shown. Figure 4.14 is the function loss between the predicted and the known output, Figure 4.15 is the regularization loss to avoid overfitting and Figure 4.16 demonstrates the total loss. The last one is the sum up of the previous two.

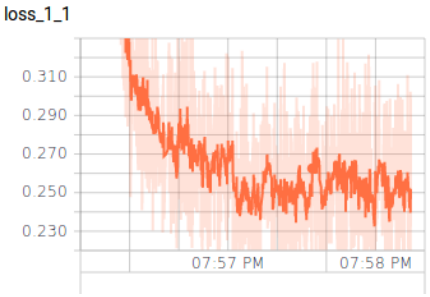


Figure 4.14 Function loss (2 UAVs and no obstacles).

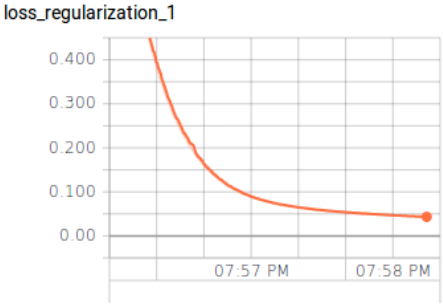


Figure 4.15 Regularization function loss (2 UAVs and no obstacles).

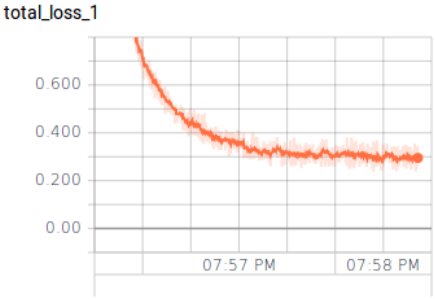


Figure 4.16 Total function loss (2 UAVs and no obstacles).

These are the scenarios researched in this paper and, while, we do not increase the complexity of the scenario, its research starts by employing a bigger and better data set with a deeper training.

In this section, a part of the optimization process, one of the most relevant pieces of information given is the velocity of the neural networks' training. If we compare the execution time of the learning between using the Matlab Neural Network Toolbox and TF, the balance falls in favour of TF, providing a faster computation. This is the advantage of using TensorFlow.

Simulation results

Once the optimal cases has been implemented giving a good learning percent, it is time to simulate them. As commented in Section 4.1, these tests will be run in the Gazebo and ROS simulator. Before that it is necessary to install Gazebo, ROS and UAL on the computer to simulate the different neural networks and see their performances. When a simulation in ROS finishes, the software saves a file named *performance_set*, where we can find the simulations that have finished and if the UAV has succeeded achieving goal position or, in contrast, it has collided. This file will be the tool to check how good the NN for the specific scenario is.

Before testing a model and during its training, we have to save its graph. Therefore, we will create a folder where the saved models are stored to use them necessary. There will be as many subfolders as different worlds are trained. In the next lines of Code 5.1, we explain the saving process for the NN for the scenario with one UAV and no obstacles, which is the first scenario that is going to be simulated. In a general case, the path to the saved model will depend on the trained scenario.

Code 5.1 Saving a graph.

```
saver = tf.train.Saver(save_relative_paths = True)
saver.save(session, "~/world_1_0/world_1_0")
```

After that, it is possible to import the complete graph with all the variables updated and restore the operations needed. In these models, the main operation is the *multilayer_model* function which returns the predicted outputs. Therefore, the next Code 5.2 is introduced in the simulator code to do this task. As we can see in Code 5.2,the *multilayer_model* function is fed with the variable *inputs*. It is the vector of inputs for each instant of the simulation. This operation returns the next output labels or, in other words, the next velocities for the reference UAV. Each iteration, a new input vector will feed *multilayer_model* function into getting new velocities. This gives us the possibility of guide an UAV in real time. The path gave to the *tf.train.import_meta_graph* class is the directory where the different trained and saved algorithms are stored to import, restore and simulate them. In the next Code 5.2, we create a session in order to evaluate the tensors of the imported graph which was previously saved.

Code 5.2 Importing and restoring a trained NN.

```
with tf.Session() as sess:
          new_saver = tf.train.import_meta_graph('~/world_1_0/world_1_0.meta')
          new_saver.restore(sess, tf.train.latest_checkpoint('~/world_1_0'))
          model_multilayer =tf.get_default_graph().get_operation_by_name('
              multilayer_model').outputs[
          new_velocity_twist = sees.run(model_multilayer, feed_dict{inputs})
```

Although we feed the model at each instant, it is important to mention, that we import and restore the NNs only once. This will produce a lower computation cost.

After this is computed, the *inputs* variable and the trained model employed will depend on the scenario that wants to be simulated. Hereafter, we will go through the different architectures created, checking and analysing their performances. It is important to know that every world is going to be simulated 100 times with their respective neural network. After that and looking into their *performance_set.csv* file, we are able to see the number of successful simulations. This give us a percent of how good is our algorithm for each world. In the simulations, the UAV will start on the floor. Then after everything is set up, it goes up starting the relevant part of the simulation. It will go through four different goal positions avoiding obstacles and then landing. Every simulation will have a randomly component, which will cause some variance in the performances. Hereafter, the term goal position and waypoint (WP), is going to be used indistinctly.

Scenario 1 UAV and 0 obstacle

To start with, we simulate the easiest scenario in the Gazebo simulator. This task allows us to check if the import and restore process was well done and if the UAV is working as it is wanted.

Once the 100 simulations are made and looking into the *performance_set.csv* file, we get the result of a 100% of favourable simulations. This means that the UAV goes from the initial point to the different WPs satisfactorily.

The next Figures are the different stretches made by a UAV through the WPs of a specific scenario. Although the movement are not perfectly optimized, the drone follows correctly its goal positions.

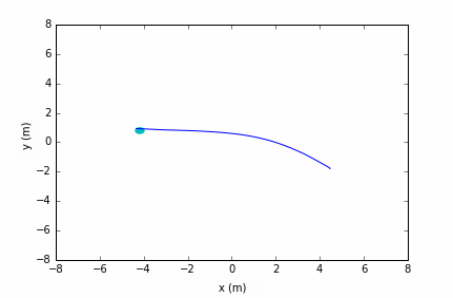


Figure 5.1 First stretch (1 UAV and no obstacles).

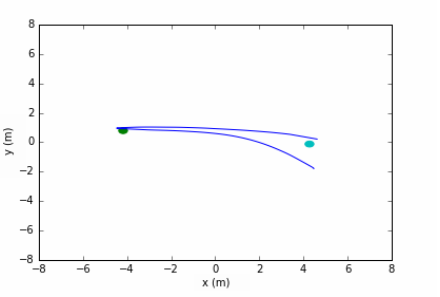


Figure 5.2 Second stretch (1 UAV and no obstacles).

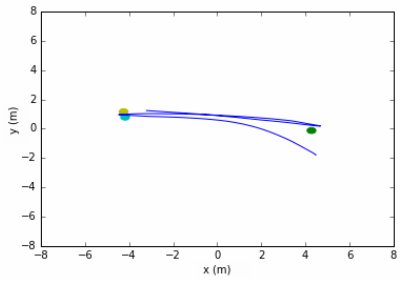


Figure 5.3 Third stretch (1 UAV and no obstacles).

These results give us the confirmation that the graph of the NN is good reused and we can go one step further introducing an obstacle.

One issue to see before starting more complex simulations, is that the time at which a new velocity is given, is around 0.2 - 0.3 seconds, staying fix independently of using ORCA or TF. This is important, in order to see if we have to train the models with TF at a different computation velocity as ORCA has provided.

Scenario 1 UAV and 1 obstacle

Now, we increase the complexity of the scenario, adding an obstacle. In the figures below, we can see how the drone goes from one point to another one avoiding colliding with the obstacle in an efficient way. The error percent of the algorithm implemented in this scenario is of 1%. This means that in 100 simulations the UAV collides only once with the obstacle. This is approximate the same result as with Matlab Toolbox instead of TF. Another relevant value of the simulations is the execution time. Here, the simulation takes around 73 seconds before it finishes. This time starts when ROS is called and finishes when the UAV landed.

Following are illustrated four different simulations of this scenario. The red circle indicates the obstacle, the blue line is the trajectory followed by the drone and the small circles, the WPs. The green, cyan, blue and purple circles are the different WPs in increasing order.

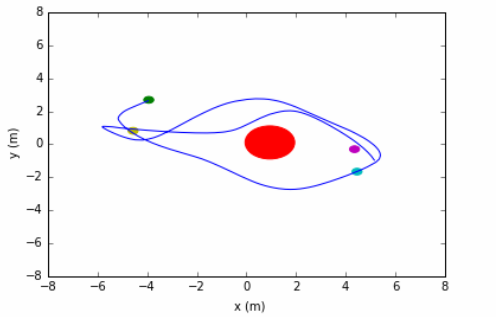


Figure 5.4 First simulation (1 UAV and 1 obstacle).

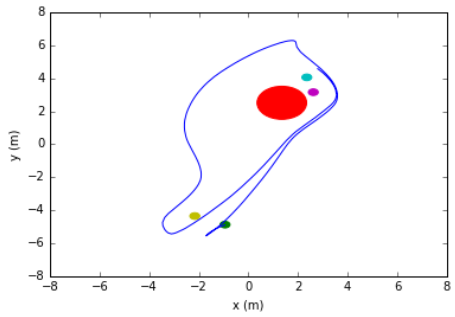


Figure 5.5 Second simulation (1 UAV and 1 obstacle).

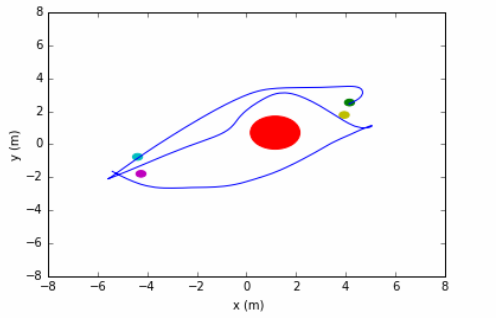


Figure 5.6 Third simulation (1 UAV and 1 obstacle).

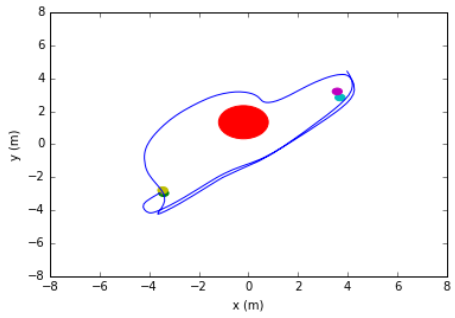


Figure 5.7 Fourth simulation (1 UAV and 1 obstacle).

In the simulations, the UAV is correctly guided in the scenario, avoiding the obstacle. Sometimes, the drone does not perceive that it has reached the WP and therefore, it flies over. This problem comes from the UAL controller which is not part of our study. In general and from the learning part, the movement of the drone could be more softly and effective. On the one hand, we could get better result with a deeper learning process, trying to avoid errors in the accuracy learning. On the other hand, the dataset employed, due to the algorithm ORCA, does not always give the best performance and makes that the neural network does not learn efficiently enough from the scenario.

Scenario 2 UAV and 0 obstacle

In this case, the scenario is a bit different and we will have two UAVs and therefore, two drones to guide. The neural networks employed to control each drone will have the same architecture, but their inputs will be different, due to that the positions are relative.

Although this scenario can be considered more difficult, the suitable results of the learning process allows to guide correctly the two UAVs with a good percent of favourable simulations. The error here based in 100 simulations is of the 27%. Although, this model can not be employed in a real system, it gives us the certain that it is possible to control UAVs in more difficult scenarios with neural networks. The time of execution of a simulation here is around 82 seconds.

The next Figures illustrate four stretches in one simulation to see how each drone try to avoid colliding with the other. It is important to mention, that the drones do not know where the other ones are. This can be a future research. In Figure 5.8, the green circle indicates the starting point of each drone and the blue one, their next WP, while the lines are the different trajectories. One stretch after, the drones try to achieve the navy blue and then, in Figure 5.10, they try to go to the red circles. The full simulation is shown in Figure 5.11.

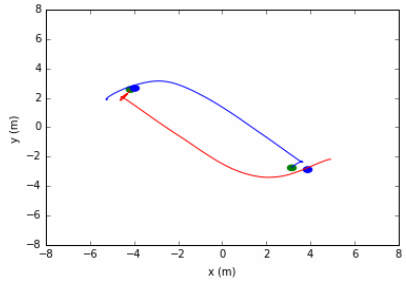


Figure 5.8 First stretch (2 UAVs and no obstacles).

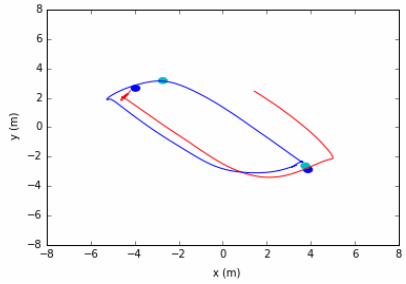


Figure 5.9 Second stretch (2 UAVs and no obstacles).

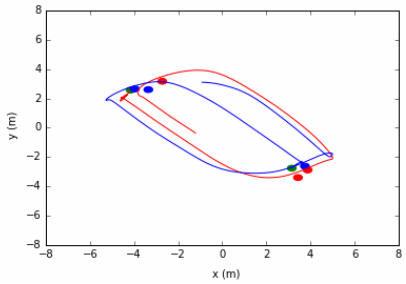


Figure 5.10 Third stretch (2 UAVs and no obstacles).

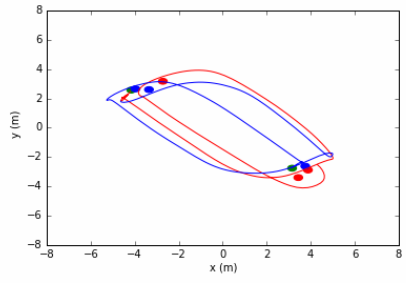


Figure 5.11 Fourth stretch (2 UAVs and no obstacles).

Now in Figure 5.12, we will show how a wrong simulation of this scenario looks. They start again in a green point and while the red trajectory follows red points, the blue one follows blue circles. Although, the drones do not directly collide trying to achieve the third WP, they pass really near and therefore the simulation is considered as unsucceeded.

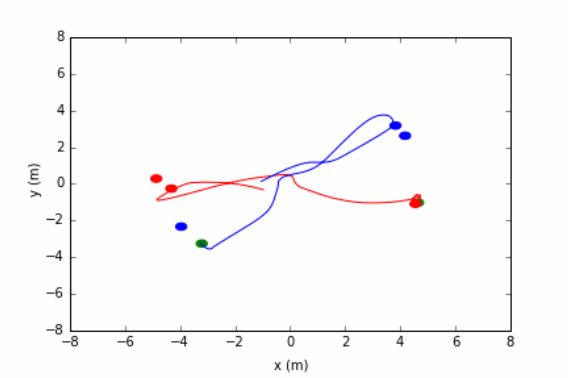


Figure 5.12 Wrong simulation (2 UAVs and no obstacles).

This type of error comes from a bad or poor training and must be eliminated in order to employ this algorithm in real systems. Some of the ways to teach better the network are:

- using a deeper and more efficiency architecture;
- using a better and bigger dataset provided by ORCA;
- employing GPUs to simulate, in order to have a faster computation.

Scenario 2 UAV and 0 obstacle without unsucceeded simulations filter

In this case, we will implement a neural network without the simulations filter, in order to check the difference between using an algorithm with and without it. This means that the neural network will also learn from simulations of the dataset where the drones have collided at some points. Although the number of simulations employed is the same, 34 simulations, the filter eliminates the wrong simulations, ending up with a dataset of 26 simulations, around 5,600 instants. In this case, we will learn from the full dataset, around 7,400 instants, in which 8 simulations are unsucceeded. The aim of this section is to compare the performances of both algorithms in order to see how affect the wrong simulations the learning process.

Therefore, the architecture implemented will be the same, with the only change of the dataset. Now, the learning accuracy is again around the 97% but the error percent in the 100 simulations decreased to the 17%. This can be a bit unintelligible, but it can be possible as going to be explained.

The filter avoids that the optimization process learns from simulations where the drones have flown to close to each other or even, have collided. This can be seen from two different point of views. On the one hand, if we use the filter, the dataset employed in the learning will always be correct, avoiding that the optimization process takes as a correct movement, something incorrect. On the other hand, in the wrong simulations, there are many instants in which the movements of the drones are correct and are important to find out due to its complexity. Normally, the unsucceeded simulations are the ones which are more difficult. Although, all simulation is considered as wrong, there is a lot of relevant information that should not be disqualify. The main problem here is that when the filter is employed, the dataset used to train the model is too poor and small, while without it, the data set is bigger. In this case, we can see that sometimes the learning of a few wrong movements are not catastrophe for the learning, but it should be avoided.

After these results, we decide to train the model again with more dataset in both cases, with and without the filter, and with the same architecture. Now the result using the filter is unconditionally better, due to the number of collided simulations is bigger and the NN without the filter learns from more wrong movements. This indicates the importance of a good dataset and a good training, avoiding generalization and a poor optimization process.

The idea in this investigation as for future researches is to create a dataset with ORCA without any collisions, independently of the randomness of the scenario. After that, the neural networks will be able to guide efficiently the drones in different scenarios.

Conclusions and future investigations

To conclude, it has been once again demonstrated how useful and interesting can be the implementation of a neural network to compute the guiding of an UAV. After several configurations, we have ended up with NNs with good "learning" percent. Then and, according to the simulations tested, we can have the certain that the drones are able to move with such as small error that they will not crash in an environment with certain complexity. Nevertheless, there are many tasks that still have to be solved in order to be able to implement this type of algorithm in real systems.

After many simulations, one of the most important reached conclusion is the importance of employing a HW and SW with a high computation power. It was relevant to carry out the simulations using the computer only for this issue. Otherwise the simulations may not work as expected. This demonstrates how much potential is necessary by the computer to simulate, with ROS and Gazebo, neural networks created by TF. Therefore, if this will be implemented in real systems with this type of algorithm, we would recommend using GPUs instead of CPUs.

In addition, it has been demonstrated how TensorFlow is a powerful tool to create neural networks, giving good results to compute deep learning algorithms. It has the needed flexibility to allow the programmer to try different configurations, reaching better performances. Nevertheless, it can also make the optimization process longer due to too many parameters to change and each modification affects a lot the result. In general, we have demonstrated how TF works and how it can be helpful in the computation of several applications.

To finish with, this paper was only an introducing to what TensorFlow and neural networks are able to do in different deep learning algorithms tasks and, in concrete, in the guiding of drones in specific environments. If we are focused in the control of an UAV with TensorFlow, there are many possibilities to explore in future researches. Following we will mention only a few of them:

- more complexity of the scenarios, adding more UAVs and obstacles;
- scenarios in 3D, increasing the number of inputs and outputs;
- implementation of deeper algorithms;
- creation of new network architectures;
- implementation of faster and more powerful software by using GPUs;
- better dataset without collisions;
- introduction to the UAV dynamics

Codes

Code 7.1 Extraction of the dataset in the simulation.

```
"""
@author: rebeca fernandez niederacher
"""

def find_pos(uav, first, last):
    try:
        start = uav.index(first) + len( first )
        end = uav.index(last, start )
        return uav[start:end]
    except ValueError:
        return ""

def find_orient(uav, first, last):
    try:
        start = uav.rindex(first) + len(first)
        end = uav.rindex(last,start)
        return uav[start:end]
    except ValueError:
        return ""

def pos_orien_vel(input_data, n_uav, UAV,lin_uav_x, lin_uav_y, goal_pos_x, goal
    _pos_y, new_lin_x, new_lin_y,pos_uav_x,pos_uav_y, position_x,position_y,lin
    _x, lin_y):
    for uav in range (1, n_uav+1):
        uav_pos = []
        uav_vel = []
        new_vel = []
        goal_position = []
        goal = False
        if uav == UAV:
            goal = True
        else :
            goal = False
        position_x.append([])
        position_y.append([])
        lin_x.append([])
        lin_y.append([])

        for  j in range (0, len(input_data['actual_UAV_%s_pose' % uav])):
```

```
            uav_pos.append(input_data['actual_UAV_%s_pose' % uav][j])
            uav_vel.append(input_data['actual_UAV_%s_vel' % uav][j])

            position_x[uav-1].append(find_pos(uav_pos[j], 'x:','\n'))
            position_y[uav-1].append(find_pos(uav_pos[j], 'y:', '\n'))
            lin_x[uav-1].append(find_pos(uav_vel[j], 'x:' , '\n'))
            lin_y[uav-1].append(find_pos(uav_vel[j], 'y:' , '\n'))

            if goal == True:
                pos_uav_x.append(find_pos(uav_pos[j], 'x:','\n'))
                pos_uav_y.append(find_pos(uav_pos[j], 'y:', '\n'))
                lin_uav_x.append(find_pos(uav_vel[j], 'x:' , '\n'))
                lin_uav_y.append(find_pos(uav_vel[j], 'y:' , '\n'))

                goal_position.append(input_data['goal_UAV_%s_pose' % uav][j])
                new_vel.append(input_data['UAV_%s_new_velocity_twist' % uav][j])

                goal_pos_x.append(find_pos(goal_position[j], 'x:','\n'))
                goal_pos_y.append(find_pos(goal_position[j], 'y:', '\n'))
                new_lin_x.append(find_pos(new_vel[j], 'x:' , '\n'))
                new_lin_y.append(find_pos(new_vel[j], 'y:' , '\n'))
```

Code 7.2 Creation of the input and output data of a neural network.

```
"""
@author: rebeca fernandez niederacher
"""
import datos
import w_definition
import pandas as pd

input_matrix = []
label = []
position_x = []
position_y = []
lin_x = []
lin_y = []
pos_uav_x = []
pos_uav_y= []
new_lin_x = []
new_lin_y = []
goal_pos_x = []
goal_pos_y = []
lin_uav_x = []
lin_uav_y = []

obs_x = w_definition.position_obs_x
obs_y = w_definition.position_obs_y
obstacle_x = []
obstacle_y = []

n_uav = w_definition.uav
n_obs = w_definition.obs
dir_input = w_definition.dir_data
num_sim = w_definition.num_sim
valid_simulation = w_definition.valid_simulation

for sim in range (0, num_sim):
    for UAV in range (1, n_uav+1):
        input_data = []
        input_data = pd.read_csv(dir_input + "/simulation_%s" % valid_
            simulation[sim] + "/uav_%s.csv" % UAV, sep=',')
        datos.pos_orien_vel(input_data, n_uav, UAV,lin_uav_x, lin_uav_y, goal_
            pos_x, goal_pos_y,new_lin_x, new_lin_y,pos_uav_x,pos_uav_y, position
            _x, position_y, lin_x, lin_y)
        for n_obstacle in range (0, n_obs):
            obstacle_x.append([])
            obstacle_y.append([])
            for len_obs in range (0, len(input_data['actual_UAV_%s_pose' % UAV])
                ):
                obstacle_x[n_obstacle].append(obs_x[sim][n_obstacle])
                obstacle_y[n_obstacle].append(obs_y[sim][n_obstacle])

for j in range(0, len(pos_uav_x)):
    input_matrix.append([])
    label.append([])
    input_matrix[j].append(float(lin_uav_x[j]))
    input_matrix[j].append(float(lin_uav_y[j]))
    for i in range (0, n_uav):
```

```
        if(float(pos_uav_x[j]) != float(position_x[i][j])):
            input_matrix[j].append(float(position_x[i][j]) -float(pos_uav_x[j]))
            input_matrix[j].append(float(position_y[i][j]) -float(pos_uav_y[j]))

        if(float(lin_uav_x[j]) != float(lin_x[i][j])):
            input_matrix[j].append(float(lin_x[i][j]))
            input_matrix[j].append(float(lin_y[i][j]))

    if (n_obs > 0):
        for num_obstacle in range (0,n_obs):
            input_matrix[j].append(float(obstacle_x[num_obstacle][j]) - float(
                pos_uav_x[j]))
            input_matrix[j].append(float(obstacle_y[num_obstacle][j]) - float(
                pos_uav_y[j]))

    input_matrix[j].append(float(goal_pos_x[j]) - float(pos_uav_x[j]))
    input_matrix[j].append(float(goal_pos_y[j]) - float(pos_uav_y[j]))

    label[j].append(float(new_lin_x[j]))
    label[j].append(float(new_lin_y[j]))

features_len = len(input_matrix[0])
simulation_len = len(input_matrix)
label_len = len(label[0])
```

Code 7.3 Filtration of the dataset and extraction of the obstacles position.

```
"""
@author: rebeca fernandez niederacher
"""
import os
import pandas as pd
import numpy as np
import csv

dir = os.getcwd()
pos_obs = []
type_world = []
archivo = []
num_sim=0

sim_num = []
error = []
valid_simulation = []
sim = []

position_obs_x = []
position_obs_y = []

pos_obs_x = []
pos_obs_y = []


uav = # Depending on the neural network to train
obs = # Depending on the neural network to train

dir_data = dir+ "/type1_Nuav%s" % uav + "_Nobs%s" % obs + "/dataset_1"
data_set = pd.read_csv(dir_data + "/performance_info.csv", sep=',')
total_sim = len(data_set['simulation_n'])

for i in range (0,total_sim):
    sim_num.append(data_set['simulation_n'][i])
    error.append(data_set['succeed'][i])
for j in range (0,total_sim):
    if(error[j] == True):
        valid_simulation.append(data_set['simulation_n'][j])

num_sim = len(valid_simulation)

for sim in range (0, num_sim):
    dir_world = dir_data + "/simulation_%s" % valid_simulation[sim]
    for archivo in os.listdir(dir_world):
        input_world=[]
        if archivo.startswith("world"):
            input_world = pd.read_csv(dir_world+ "/world_definition.csv", sep
                =',')
            if (obs > 0):
                pos_obs= np.matrix(input_world['obs_pose_list_simple'][0])
    if (obs > 0):
        for obstacle in range (0,obs):
            offset = 3*obstacle
            pos_obs_x.append(float(pos_obs.T[0+offset]))
```

```
            pos_obs_y.append(float(pos_obs.T[1+offset]))

if (obs > 0):
    position_obs_x = np.reshape(pos_obs_x, (num_sim,obs))
    position_obs_y = np.reshape(pos_obs_y, (num_sim,obs))
```

Code 7.4 Neural network trained for scenario with 1 UAV and 1 obstacle.

```
"""
@author: rebeca fernandez niederacher
"""
import os
import tensorflow as tf
import w_definition
import datos_entrada
import numpy as np
from sklearn.metrics import accuracy_score

n_csv = w_definition.uav
n_obs = w_definition.obs
num_label = datos_entrada.label_len
input_len = int(1*datos_entrada.simulation_len)
num_features = datos_entrada.features_len

batch_data =[]
batch_labels = []

train_size = int(0.7*input_len)
valid_size = int(0.15*input_len)

tf.app.flags.DEFINE_integer('num_epochs', 800,
                          'Number of examples to separate from the training '
                          'data for the validation set.')

tf.app.flags.DEFINE_float('learning_rate', 0.006, 'Initial learning rate.')

tf.app.flags.DEFINE_integer('train_size', train_size, 'set of training data')

tf.app.flags.DEFINE_integer('valid_size', valid_size, 'set of validation data')

tf.app.flags.DEFINE_integer('BATCH_SIZE', 20, 'Must divide evenly into the
    dataset sizes.') #ntes 6

tf.app.flags.DEFINE_integer('n_hidden_1', 30, 'Number of neuron in layer 1')

tf.app.flags.DEFINE_integer('n_hidden_2', 30, 'Number of neuron in layer 2')

tf.app.flags.DEFINE_integer('n_hidden_3', 30, 'Number of neuron in layer 3')

FLAGS = tf.app.flags.FLAGS

def randomize(dataset, labels):
    permutation = np.random.permutation(len(labels))
    shuffled_dataset = dataset[permutation, :]
    shuffled_labels = labels[permutation, :]
    return shuffled_dataset, shuffled_labels

data_input = datos_entrada.input_matrix
labels_input = datos_entrada.label

data_input = np.matrix(data_input, np.float32)
labels_input = np.matrix(labels_input, np.float32)
```

```
error = False
num_epochs = FLAGS.num_epochs
learning_rate = FLAGS.learning_rate
BATCH_SIZE = FLAGS.BATCH_SIZE
train_size = FLAGS.train_size
valid_size = FLAGS.valid_size
n_hidden_1 =FLAGS.n_hidden_1
n_hidden_2 =FLAGS.n_hidden_2
n_hidden_3 =FLAGS.n_hidden_3

if (train_size>input_len):
    error = True
    print ("ERROR!")
    print ()
    print ("TRAINING SIZE BIGGER THAN THE COMPLETE SIMULATION SET. YOU SHOULD
        MAKE IT SMALLER")
    print ()
    print ("training size:")
    print (train_size)

dataset, labels = randomize(data_input,labels_input)
train_data = dataset[0:train_size, :]
train_labels = labels[0:train_size, :]
valid_data = dataset[train_size: valid_size+train_size, : ]
valid_labels = labels[train_size: valid_size+train_size, :]
test_data= dataset[valid_size+train_size : input_len, :]
test_labels= labels[valid_size+train_size : input_len, :]

beta1=0.005
beta2 = 0.001

graph = tf.Graph()
with graph.as_default():

    tf_train_dataset = tf.placeholder(tf.float32, shape=(None, num_features),
        name='input_matrix')
    tf_train_labels = tf.placeholder(tf.float32, shape=(None, num_label))
    tf_valid_dataset = tf.constant(valid_data)
    tf_test_dataset = tf.constant(test_data)

    dev_1 = 0.7
    dev_2 = np.sqrt(2/n_hidden_1)
    dev_3 = np.sqrt(2/n_hidden_2)
    dev_out = np.sqrt(2/n_hidden_3)

    weights = {
        'h1': tf.Variable(tf.random_normal([num_features, n_hidden_1], stddev =
             dev_1, dtype= np.float32)),
        'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2], stddev =
            dev_2, dtype= np.float32)),
        'h3': tf.Variable(tf.random_normal([n_hidden_2, n_hidden_3], stddev =
            dev_3, dtype= np.float32)),
        'out': tf.Variable(tf.random_normal([n_hidden_3, num_label], stddev =
            dev_out, dtype= np.float32))
        }

    biases = {
```

```
        'b1': tf.Variable(tf.zeros([n_hidden_1], dtype= np.float32)),
        'b2': tf.Variable(tf.zeros([n_hidden_2],dtype= np.float32)),
        'b3': tf.Variable(tf.zeros([n_hidden_3],dtype= np.float32)),
        'out': tf.Variable(tf.zeros([num_label],dtype= np.float32))
    }

    def multilayer_perceptron(x):
        # Hidden fully connected layer with n_hidden_1 neurons
        layer_1 = tf.nn.tanh(tf.add(tf.matmul(x, weights['h1']), biases['b1']))
        # Hidden fully connected layer with n_hidden_2 neurons
        layer_2 = tf.nn.tanh(tf.add(tf.matmul(layer_1, weights['h2']), biases['
            b2']))
         # Hidden fully connected layer with n_hidden_3 neurons
        layer_3 = tf.nn.tanh(tf.add(tf.matmul(layer_2, weights['h3']), biases['
            b3']))
        # Output fully connected layer with a neuron for each class
        out_layer = tf.nn.tanh(tf.matmul(layer_3, weights['out']) + biases['out
            '])

        return (out_layer)


    logits = multilayer_perceptron(tf_train_dataset)
    loss = tf.losses.mean_squared_error(labels=tf_train_labels, predictions =
        logits)

    optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
    train_prediction = logits

    valid_prediction = multilayer_perceptron(tf_valid_dataset)
    test_prediction = multilayer_perceptron(tf_test_dataset)

def accuracy(predictions, labels):
    percent = accuracy_score(np.argmax(labels,1), np.argmax(predictions,1))*100
    return percent

with tf.Session(graph=graph) as session:

    tf.global_variables_initializer().run()
    print("Initialized")
    for step in range((num_epochs * train_size // BATCH_SIZE)):
        offset = (step * BATCH_SIZE) % (train_size-BATCH_SIZE)
        batch_data = train_data[offset:(offset + BATCH_SIZE), :]
        batch_labels = train_labels[offset:(offset + BATCH_SIZE), :]
        feed_dict = {tf_train_dataset: batch_data, tf_train_labels: batch_
            labels}
        _, l, l_regu, predictions = session.run([optimizer, loss, loss_regu,
            train_prediction], feed_dict=feed_dict)
        if (step % 2000 == 0):

            print("Minibatch loss at step %d: %f" % (step, l))
            print ("loss regularization at step %d: %f" % (step, l_regu))
            print("Minibatch accuracy: %.1f%%" % accuracy(predictions, batch_
                labels))
            print("Validation accuracy: %.1f%%" % accuracy(valid_prediction.eval
                (), valid_labels))
```

```
    print("Test accuracy: %.1f%%" % accuracy(test_prediction.eval(), test_
        labels))
```

List of Figures

List of Codes

References

Analytics, P., Networks, N., & Buduma, B. N. (2000). Data Science 101 : Preventing Overfitting in Neural Networks. , 3–5. Retrieved from https://www.kdnuggets.com/2015/04/preventing-overfitting-neural-networks.html/2

Besbes, A. (2017). *Understanding deep Convolutional Neural Networks with a practical use-case in Tensorflow and Keras.* Retrieved 2017-12-04, from https://ahmedbesbes.com/understanding-deep-convolutional-neural-networks-with-a-practical-use-case-in-tensorflow-and-keras.html

Brownlee, J. (2016). *Supervised and Unsupervised Machine Learning Algorithms - Machine Learning Mastery.* Retrieved 2017-11-27, from http://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/

Davis, A. L., & Keller, R. M. (1982). Data-flow graphs. *Computer*, 26–41. Retrieved from http://bears.ece.ucsb.edu/research-info/DP/dfg.html

Dean, J., & Monga, R. (2015). *TensorFlow - Google's latest machine learning system, open sourced for everyone.* Retrieved 2017-11-26, from https://research.googleblog.com/2015/11/tensorflow-googles-latest-machine{_}9.html

Dean, J., Monga, R., Mital, P. K., TensorFlow, Davis, A. L., Keller, R. M., … Louradour, J. (2000, sep). *TensorFlow* (Vol. 1). IEEE Comput. Soc. Retrieved 2017-11-26, from http://ieeexplore.ieee.org/document/6981034/https://www.jetbrains.com/pycharm/https://www.tensorflow.org/http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/http://www.lavanguardia.com/tecnologia/20170716/42411 doi: 10.1109/ICDAR.2003.1227801

Dean James. (2014). *What's the Difference Between AI and Machine Learning?* Retrieved 2017-11-29, from http://www.machinedesign.com/industrial-automation/what-s-difference-between-ai-and-machine-learning

Estimators | TensorFlow. (2000). Retrieved 2017-11-26, from https://www.tensorflow.org/programmers_guide/estimators

Google. (2018). *Simple Audio Recognition | TensorFlow.* Retrieved 2017-11-28, from https://www.tensorflow.org/versions/master/tutorials/audio{_}recognition

Google LLC. (2018). *Graphs and Sessions.* Retrieved 2017-11-27, from https://www.tensorflow.org/programmers{_}guide/graphs

Installing TensorFlow on Mac OS X. (2000). Retrieved 2017-12-05, from https://www.tensorflow.org/install/

Karn, U. (2016). A Quick Introduction to Neural Networks. *The Data Science Blog*. Retrieved from https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/

Machine Learning y Deep Learning: cómo entender las claves del presente y futuro de la inteligencia artificial. (2000). Retrieved 2017-11-29, from https://www.xataka.com/robotica-e-ia/machine-learning-y-deep-learning-como-entender-las-claves-del-presente-y-futuro-de-la-inteligencia-artificial

Mccrea, N. (2013). *A Machine Learning Introductory Tutorial with Examples | Toptal.* Retrieved from https://www.toptal.com/machine-learning/machine-learning-theory-an-introductory-primer

Mircrosoft SQL Server. (2012). Training and Testing Data Sets. *MSDN library*, 2014–2016. Retrieved from https://technet.microsoft.com/en-us/library/bb895173(v=sql.110).aspx

Mital, P. K. (2016). Session 1 - Introduction to Tensorflow. , 2–5. Retrieved from https://www.tensorflow.org/mobile/mobile{_}intro

Ongsulee, P. (2017). Artificial Intelligence, Machine Learning and Deep Learning. , 1–11. Retrieved from https://spin.atomicobject.com/2014/06/24/gradient-descent-linear-regression/ http://spin.atomicobject.com/2014/06/24/gradient-descent-linear-regression/

Parzych, D. (2017). *Artificial Intelligence vs. Machine Learning vs. Deep Learning - DZone AI.* Retrieved 2017-11-29, from https://dzone.com/articles/the-differences-between-ai-and-ml

Pham, V., Bluche, T., Kermorvant, C., & Louradour, J. (2014, sep). Dropout Improves Recurrent Neural Networks for Handwriting Recognition. In *2014 14th international conference on frontiers in handwriting recognition* (pp. 285–290). IEEE. Retrieved from http://ieeexplore.ieee.org/document/6981034/ doi: 10.1109/ICFHR.2014.55

¿Por qué regala Google el 'software' del que depende su futuro? (2000). Retrieved 2017-12-22, from http://www.lavanguardia.com/tecnologia/20170716/424112953155/google-alphabet-aprendizaje-automatico-tensorflow.html

PyCharm: Python IDE for Professional Developers by JetBrains. (2000). Retrieved 2018-03-13, from https://www.jetbrains.com/pycharm/

Python. (2014). *Download Python | Python.org.* Retrieved from https://www.python.org/download/

Python, C. (2017). Image Recognition and Object Detection _ Part 1 _ Learn OpenCV.pdf. , 1–18. Retrieved from https://www.learnopencv.com/image-recognition-and-object-detection-part1/

Python Programming Tutorials. (2000). Retrieved 2017-11-26, from https://pythonprogramming.net/tensorflow-introduction-machine-learning-tutorial/

Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs – WildML. (2000). Retrieved 2018-03-12, from http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

Simard, P., Steinkraus, D., & Platt, J. (2000). Best practices for convolutional neural networks applied to visual document analysis. In *Seventh international conference on document analysis and recognition, 2003. proceedings.* (Vol. 1, pp. 958–963). IEEE Comput. Soc. Retrieved from http://ieeexplore.ieee.org/document/1227801/ doi: 10.1109/ICDAR.2003.1227801

Started, G., & To, H. (2016). MNIST For ML Beginners. , 1–11. Retrieved from https://www.tensorflow.org/get{_}started/mnist/beginners

Techopedia. (2015). *What is the Internet of Things (IoT) - Definition from Techopedia.* Retrieved 2017-12-13, from https://www.techopedia.com/definition/28247/internet-of-things-iot

TensorFlow. (2015). TensorFlow Architecture. Retrieved from https://www.tensorflow.org/extend/architecture

Tensorflow. (2017). *API Documentation.* Retrieved 2017-11-26, from https://www.tensorflow.org/api{_}docs/

TensorFlow. (2017). *TensorBoard: Visualizing Learning.* Retrieved 2017-12-05, from https://www.tensorflow.org/get{_}started/summaries{_}and{_}tensorboard

TensorRec: A Recommendation Engine Framework in TensorFlow. (2000). Retrieved 2017-11-27, from https://hackernoon.com/tensorrec-a-recommendation-engine-framework-in-tensorflow-d85e4f0874e8

Tensors | TensorFlow. (2000). Retrieved 2017-11-28, from https://www.tensorflow.org/programmers{_}guide/tensors

Top Five Use Cases of TensorFlow. (2000). Retrieved 2017-11-27, from https://www.exastax.com/deep-learning/top-five-use-cases-of-tensorflow/

Wongsuphasawat, K., Smilkov, D., Wexler, J., Wilson, J., Fritz, D., Krishnan, D., … Wattenberg, M. (2000). Visualizing Dataflow Graphs of Deep Learning Models in TensorFlow. Retrieved from https://idl.cs.washington.edu/files/2018-TensorFlowGraph-VAST.pdf