Escuela Técnica Superior de Ingenieros
Industriales y de Telecomunicación
UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Máster

Explorando redes neuronales convolucionales para reconocimiento de objetos en imágenes RGB

Exploring Convolutional Neuronal Networks for RGB-based Object Recognition

To achieve the title of

Master in Electronics for Embedded Systems

Author: Matteo Pera
Supervisors: Prof. Pablo Pedro Sanchez Espeso
Prof. Edgar Ernesto Sanchez Sanchez

September - 2020

Contents

List of Figures

Introduction

1.1 Context

In this historical period of technological innovations many tools are becoming fully automatized, consider for example public transport, cars, robotic arms in assembly lines and many others.
Nowadays it seems like the trend is to use computers to replace the human work. It may seem foolish, trying to replace ourselves with machines, actually this path can open a new world of opportunities because people will be needed for designing those computers and worrying about their maintenance.

However the real reason behind this current of thought is that the tools which are being developed work better than human beings, not only in terms of performance and time but also under the safety perspective.

A self driving car will never have an accident because of tiredness or distraction, a robotic surgeon doesn't have trembling hands and so on. People are replacing themselves in all those tasks that can endanger lives.

(a) Self Driving Car

(b) Drone

Drones are getting increasingly common with several fields of employments, for instance photographers and video makers use them during their shooting to obtain images from unusual point of views, drone races are pretty popular as entertainment,

even the police use those ones for patrolling cities.
They have a great potential in term of safety, they can be used to research a missing person or carry out deliveries for people with reduced mobility just to cite a couple of examples. Therefore in the same way cars need no more a pilot to avoid gross errors during the drive, drones will be soon fully automatized too.

1.2 Motivation

Drones already implement cameras to take images and videos or just to allow the pilot to see where it is and where it is going. This factor is the key to achieve an autonomous conduct as for the self driving cars.

Applying Machine Learning (ML) algorithms for image analysis, it is possible to examine the environment and recognize the surrounding objects, in such a way that the device implementing the code is able to take decisions about the next action. Thus a drone can decide whether to stop, go ahead, steer or go back by itself avoiding collisions and reaching its goal (for example, pick up a packet).
These algorithms are so sophisticated that could recognize objects in an image equal or better than a person. This feature combined with the faster reaction time of a computer explains the relatively good performance and the high reliability, of these systems, moreover why self driving is such a hot topic.

Self driving drones can be applied in a wide range of applications, one practical case can be package deliveries. Imagine to simply set the coordinates for a consignment on a computer, then a drone can autonomously take off with the package to send and goes around the city without accidents because it can see the surroundings and taking decisions faster than a pilot, arriving to its destination in complete safety.
Moreover entrusting this type of task to a drone rather than a person is converted into a reduction of traffic on the streets with the consequence of a decrease in road crashes risk.

1.3 Goal

The aim of this project is to produce a code for the object recognition based on a Machine Learning algorithm, to be implemented on an embedded systems so it can identify a certain number of objects indicated on a standard dataset. In this project, the used dataset is the Washington RGB [7].

The selected algorithm for object recognition uses a CRNN [8], Convolutional Recursive Neuronal Network. This type of network takes a pre-trained Convolutional Neural Network (CNN) [9] combined at the end to a Recursive Neural Network (RNN) [10] to obtain a fine grained structure. All the aspect of this system will be discussed in section 3.1. As for a lot of problems often the hybrid solutions come out better than the single elements that generated them, so it is in this case where two different types of networks as CNN and RNN are joined in a single architecture which takes the best from both.
In order to achieve the desired objective of automatic object recognition the work is divided in:

- Brief study of the existing deep learning techniques, chapter 2
- Selection of a network topology, section 3.1
- Training the Network with the Washington RGB Object Dataset, section 3.2.1
- Implement the weights obtained from training in a propagation code, section 3.2.2
- Compare performance between different RNNs, section 3.3
- Consider all the possible solutions selecting the most suitable one, chapter 4

Machine Learning

2.1 Definition

Machine Learning is kind of algorithm that is based on the concept of **Learning from Data** [11], in which the algorithm improves automatically through experience [12]. Computer obtains an empirical knowledge thanks to the data that it receives as input. The quality and number of data determines the reliability of the code; thus the true potential of this approach lies on the choose of the dataset and how it is organized.

2.1.1 Concept of learning

Humans and other animals can display behaviours that are labeled as intelligent by learning from experience, they learn from their past. This process is divided in three steps: **remember**, **adapting** and **generalising** [11].
A smart person facing an obstacle starts thinking if he has ever coped this problem, if he knows the situation, he will make the same decision which last time has revealed correct or he will change the approach in case of a previous failure. If he ends up in a new scenario he will try a solution which has worked in similar conditions adapting his actions to the circumstances. At the end he draws conclusion for the next time expanding his knowledge with new informations.

Machine learning is about bringing this improvement procedure into the digital world, its power is based on the fact that computers can store a huge amount of data which are used as meaningful examples during the learning process.
The number of available cases mixed with the speed analysis of the virtual system guarantee great results.

2.1.2 Types of Machine Learning

Computers learn from the input data which provide them informations about practical scenarios but there are two important questions:

- **How does the computer know whether it is getting better or not?**
- **How does it know how to improve?**

There are several different possible answers to these questions and they produce different types of Machine Learning [11]:

Supervised Learning: it is the easiest and most popular method for ML, during the training phase the code is fed with input data coupled with their answers called labels. After a certain number of examples the computer begins to understand the relation between input and output, it records what it learns setting some parameters, generalizing its logical path, as result when new stimuli without responses occur it is able to handle them.
At the end the fully trained network is able to make prediction, guess the labels, for data it has never seen before.

Unsupervised Learning: This approach doesn't present labels for the input, here the algorithm is provided with a massive quantity of data and tools to understand their properties. The interesting aspect is that an overwhelming majority of data in this world is unlabeled and having an intelligent algorithms that can take terabytes and terabytes of unlabeled data and make sense of it is a great resource. Because of this characteristic this method is defined "data-driven" [1].

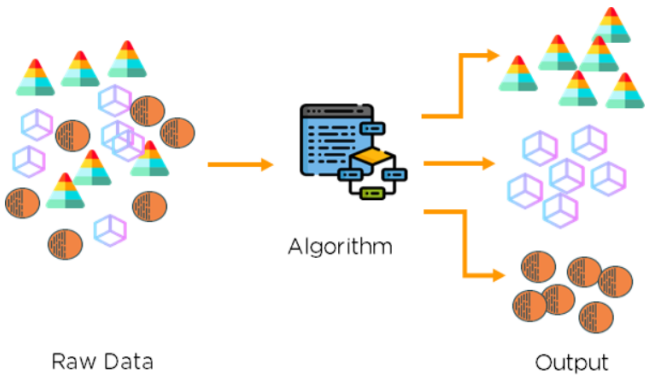


Figure 2.1: Unsupervised Learning Schematic [1].

Reinforcement Learning: This solution is a trade-off between the previous ones, it balances the exploiting of current knowledge of Supervised Learning with the exploring unknown answers proper of the Unsupervised Learning. What does it means? It tries out different possibilities until it gets the correct answer. It is called "learning with a critic" because the monitor scores the answer, but it does not suggest improvements [11].

Evolutionary Learning: This ML exploits Evolutionary Algorithms, which take inspiration from nature. Animals evolve in a specific way, during the reproduction sometimes a random genetic changing appears, this modification can be an improvement or a worsening. In case of positive change the animal survives and carries on this variation with its offspring, otherwise it perishes and its mutation is discarded. Evolutionary Learning works as the biological evolution, it starts from a solution then after some casual alterations it reaches a better result discarding the loosing variations.

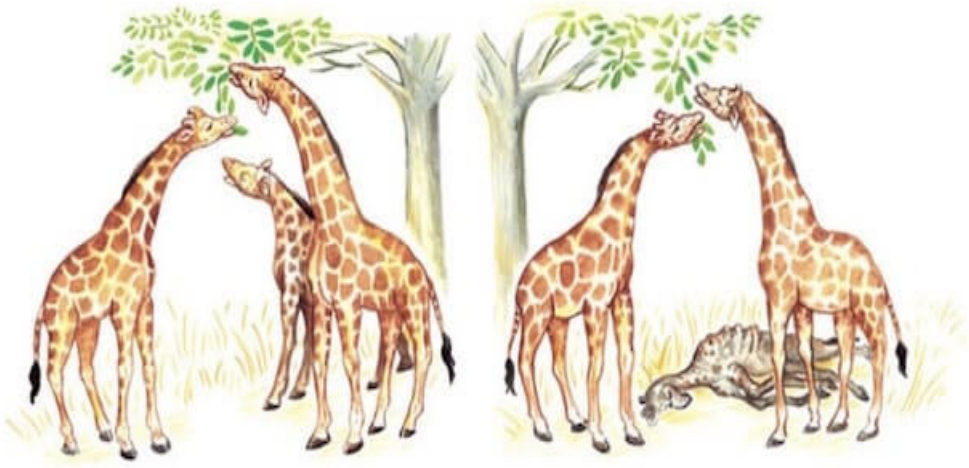


Figure 2.2: Darwin's Evolution Theory on Giraffes [2].

2.1.3 Machine Learning Concepts

This project is based on a Supervised Learning, it is simpler than the Evolutionary or the Reinforcement ones and it requires a smaller dataset than the ones for the Unsupervised Learning.
Considering the project purpose, that is object recognition, several concepts have to be introduced :

- **Dataset:** The key phase in Machine Learning development is the selection of the dataset, that can significantly ease or worsen the learning process. A dataset includes a set of images and labels that identifies the image category. An important aspect to consider for a Supervised Learning it is that not only the input data have to be prepared but labels too. Fortunately several datasets are available to be downloaded. This is an advantage because the size of the data collection is a critical point, because more data input means better performance but also longer computational time so it is important to find a trade-off. For example, ImageNet is a huge visual database designed for visual object recognition software. More than 14 million images are stored with label annotations about which object is contained in each picture. It is divided in more than 20'000 categories, every category includes several hundreds of images of the same object such as "ball" or "food can" [13].

- **Pre-trained algorithms:** Since 2010 the founders of ImageNet run an annual contest, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), so it is possible to access to a great resource of top quality algorithms that have been developed by the contenders and trained with the ImageNet dataset. This project will use a pre-trained CNN and a RNN. The selected pre-trained CNN algorithm is the runner-up of ILSVRC in the 2014, the VGGNet. Also the winner of the 2012 competition, AlexNet, was taken in consideration but the former has better results. This project also uses a RNN, Recursive Neural Network that can be set with different numbers of layers depending on the required yield. These networks will be commented on chapter 2.

- **Features:** Features are the basic building blocks for object recognition, setting the characteristics in which the algorithm is interested. This is a crucial point because they are the discriminant factors to identify a target. One single

information between shape, size and colour allows to discard a large amount of options lighten the computations.
These parameters are decided depending on the object classes present in the dataset, the final code will make predictions according to the features extracted from the input.

- **Model:** To get an algorithmic code able to handle complex data as the ones present in image processing, speech analysis and other human-like applications it is necessary a higher effort in comparison to a common Supervised Learning. The solution is *Deep learning* (DL). This is a class of Machine Learning techniques that exploit many layers of non-linear information processing for supervised or unsupervised feature extraction and transformation, for pattern analysis and classification. It consists of many hierarchical layers to process the information in a non-linear manner, where some lower-level concept helps to define the higher-level concepts [14].
  DL offers several options to work with as Autoencoders, Deep Belief Net, Convolutional Neural Networks, Recurrent or Recursive Neural Networks.
  In this design it is implemented an hybrid model between a pre-trained Convolutional and Recursive Network to guarantee better performances.

- **Training:** This is the phase where the computer learns how to take decisions based on the received inputs and its labels. Considering that the CNN implementation is already trained, in this project this phase interests only the RNN. This part takes a very long time interval depending on the dataset and its organization. For this reason it is important to find a fair balance between the number of used inputs and the wanted accuracy. The best performance could require an unfeasible computation time or resources but a code prepared with a short training dataset could be useless.

- **Evaluation:** It is the final step before the network is ready to be implemented in a device. It consists in testing the program with new images that it has never seen before, they are not present in the training records, to verify its reliability. This is the step that evaluates the worth of all the previous efforts.

The development of a Deep Learning based application normally requires several steps:

1. Select a dataset that includes images of the target categories or objects to recognize. The dataset also includes images that are not related with the target categories.

2. Select the network model that will be used to recognize the objects. There are pre-trained models that could be extended to recognize new object categories.

3. Train the selected network with the dataset.

4. Validate the network results and provide the network weights (network configuration parameter).

5. Implement the network in the application with the weights that were obtained in the previous step.

2.1.4 Noise

All Machine Learning algorithms try to find a mathematical function that correlates all the input values x_i with their response y_i. Unfortunately, during the training, Supervised Learning is sensible to two common problems called **Overfitting** and **Underfitting**.
Consider the following starting dataset:

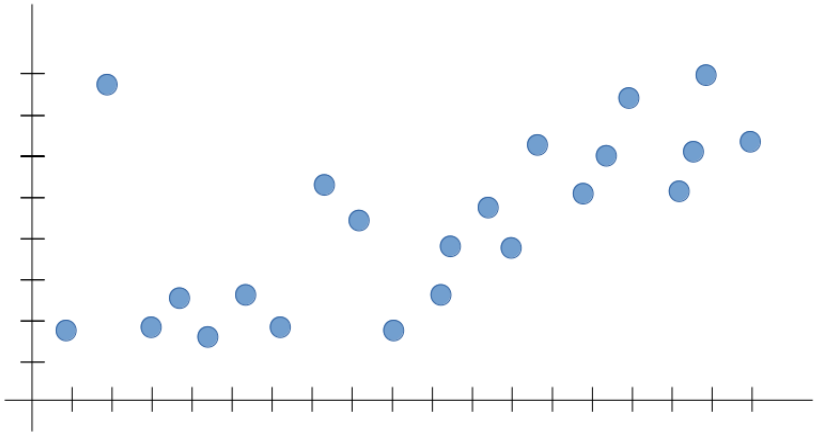


Figure 2.3: Data Samples

To find a mathematical function able to describe exactly all those data could be hard. For example, Linear Regression technique allows to map numeric inputs to numeric outputs, fitting a line into the data points. Training the Linear Regression model in the example is all about minimizing the total distance (i.e. cost) between the line to fit and the actual data points. This goes through multiple iterations until a relatively "optimal" configuration of the line within the data set occurs [15].
A possible solution based on Linear Regression could be:

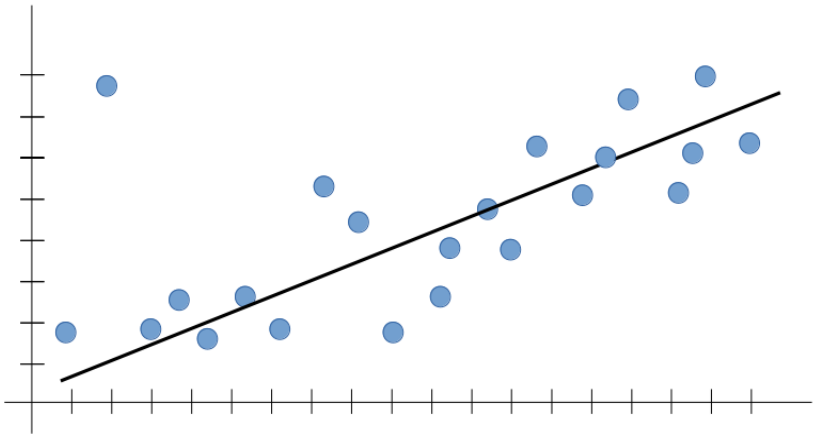


Figure 2.4: Good Linear Regression

Overfitting arise when, attempting to achieve better performance, the training runs for too much time thus the lines go to "touch" all the points ending in an excessive characterization. In this case the noisy data are not ignored and this produces a poor data generalization and a low prediction capability.

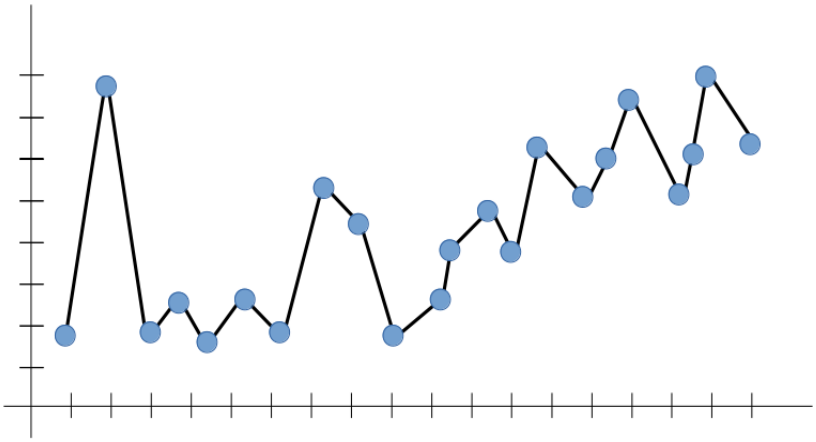


Figure 2.5: Overfitting representation

Underfitting occurs when the training runs for too little time, this lead the model to not learn enough patterns and does not get the dominant trend. This poor development results in low generalization and unreliable predictions [15].

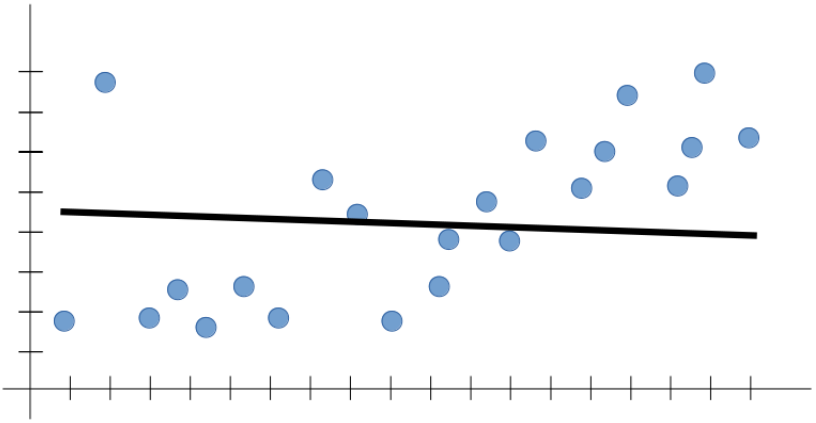


Figure 2.6: Underfitting representation

The Underfitting can be avoided simply by increasing the training time but this increases the risk of Overfitting. Luckily there are two common solutions that allow to define the function for all the necessary period without problems. They are called *Cross Validation* and *Early Stopping* but before to enter in details with them it is necessary to understand the database organization.

2.2 Dataset

2.2.1 Data Splits

As it was previously commented, the information provided to the algorithm for for training is a crucial point. In order to improve results, the dataset is usually divided in three groups: *Train, Validation* and *Test*.
Train dataset is the actual part used to educate the network, while the Validation one is used not to improve but to monitor at regular intervals the state of the process learning. Algorithms work even without the latter but it is an essential step to achieve a reliable result.
Test dataset provides the gold standard used to evaluate the model. It is only used once the training is completed. It contains carefully sampled data which spans the various classes that the model would face, when used in the real world.

Just to summarize, these data sets can be described as follows:

–**Training Dataset:** The sample of data used to fit the model.

–**Validation Dataset:** The sample of data used to provide an unbiased evaluation of a model fit on the training dataset while tuning model parameters.

–**Test Dataset:** The sample of data used to provide an unbiased evaluation of a final model fit on the training dataset [16].

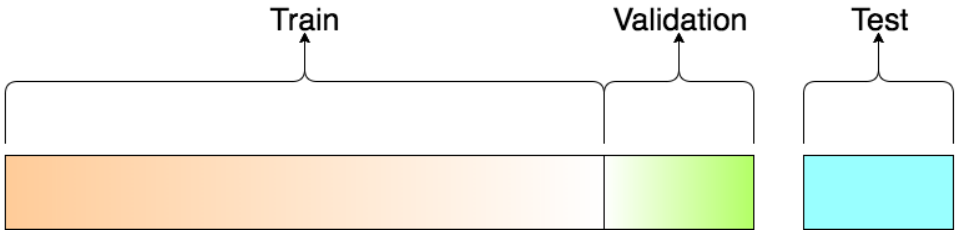


Figure 2.7: Dataset splits

How to split the available values between the parts is a delicate decision, a common way to proceed is starting with a first separation between Train and Test, then (keeping asides the Test) isolate a part of the Train to get the Validation. The

Train should be always the most conspicuous part and usually the splitting ratio are 50:25:25 or 60:20:20 depending of the resources availability.

2.2.2 Cross-Validation & Early Stopping

Cross Validation

It is a strong preventative measure against *Overfitting* for Supervised Learning. It is designed to consider just Train and Test fragments but associate a portion of the dataset to Validation part doesn't affect its benefits. The idea is to use initial data to generate multiple mini train-splits and use them to tune the model. It is also called K-fold Cross-Validation because the inputs are partitioned in k subset, called folds or bins. Then the algorithm is iteratively trained on the k-1 folds while the remaining one is used as test set, called holdout fold [17].
The k iterations require for sure more computational time but it is an affordable cost with respect to the possibility to train and test with the entire dataset.

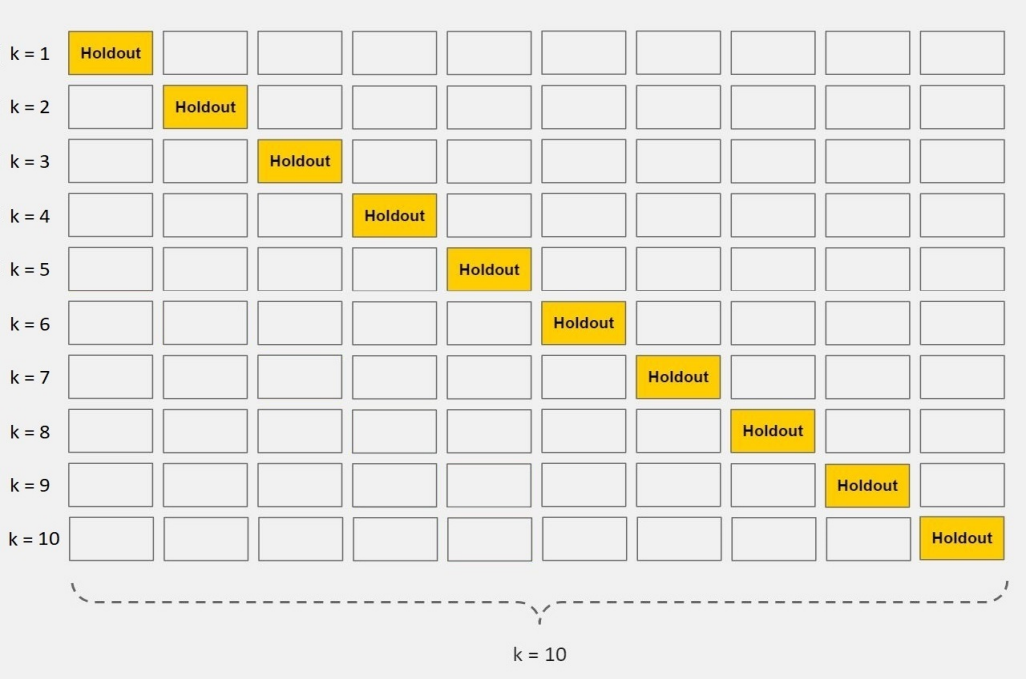


Figure 2.8: 10-folds Cross Validation

Early Stopping

A correct comprehension of Early Stopping requires the definition of several terms:

–Hyperparameter: It is a ML feature used to supervise the learning process, it is different to a common parameter which has a value derived by the code education.

–Batch: The training set is divided in a number of batch that are a collection of data. Batch size is a hyperparameter that defines the number of samples to work through before updating the internal model parameters. It must be greater or equal to one and lower or equal to the number of available samples.

–Epoch: During the training it is possible to iterate multiple times over the same given inputs. The number of epochs is a hyperparameter which defines the number of times the learning algorithm will work through the entire training dataset. One epoch means that each sample has had an opportunity to update the internal model parameters once. The number of epochs is traditionally large, often hundreds or thousands, allowing the learning algorithm to run until the error from the model has been sufficiently minimized. Each of them is comprised of one or more batches [18].

Once these concepts are consolidated it is possible to answer to the question "What is Early Stopping?".
The idea is pretty simple:

- Split initial data into training, validation and test sets
- At the end of each epoch (or, every N epochs):
    - evaluate the network performance on the validation set
    - if the network outperforms the previous best model, save a copy of the network at the current epoch
    - If the error on the validation set starts to increase, the process is arrested because the computer stopped learning from the function and it is getting wrong informations from the noise.
- Take as final model the one that has the best validation set performance [19]

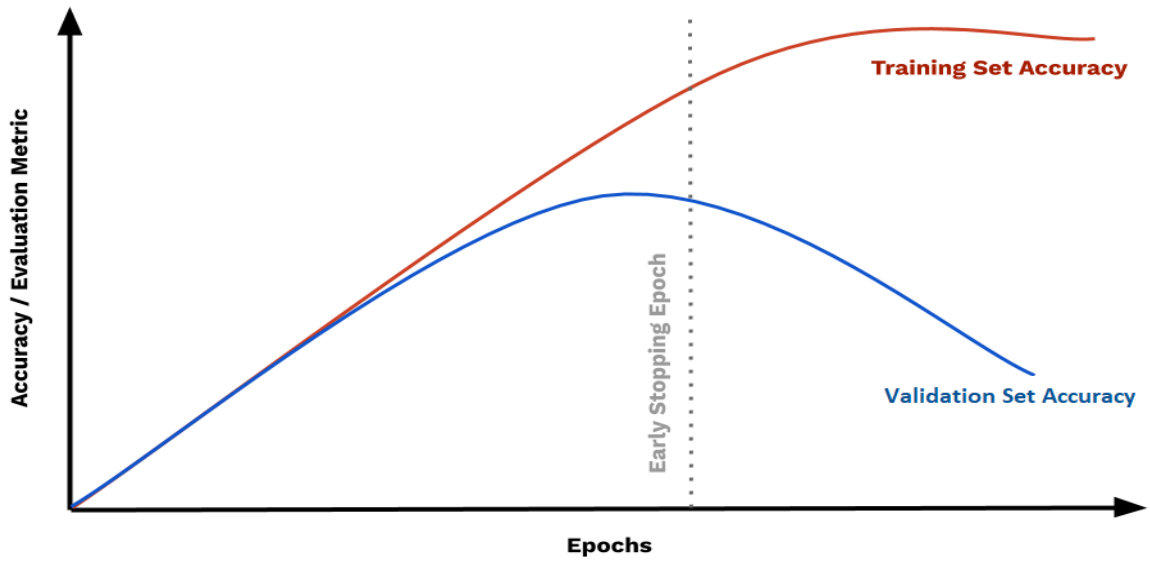


Figure 2.9: Early Stopping representation

Figure 2.9 depicts the accuracy trend. It shows that, increasing the number of epochs means an improvement of the Training and Validation sets accuracy. However, after a certain number of iterations, the Validation accuracy starts to decrease and its curve falls down. This is the sign that announces the beginning of the Overfitting. Exiting from the cycle in this moment it guarantees the best knowledge without losing the generalizing capability.

2.2.3 Washington RGB

The Washington RGB Object Dataset includes 300 common household objects, which are organized into 51 categories arranged using WordNet hypernym-hyponym relationships (similar to ImageNet). This dataset was recorded using a Kinect style 3D camera that records synchronized and aligned 640x480 RGB images at 30 Hz. Each object was placed on a turntable and video sequences were captured for one whole rotation. For each of them there are 3 video sequences, all of these recorded with the camera mounted at a different height so that the target is viewed from different angles with the horizon.

In many datasets there is no additional partition beyond the categories, for example a class for dog images contains hundreds pictures from many different dogs and there is no way to tell whether two images contain the same dog.

This dataset, instead, is organized into both categories and instances. Each category is further divided in several instances as in the case of the "soda can" class that is split in physically unique type as Pepsi, Mountain Dew and others.
In the next picture there are some example objects that have been segmented from the background [7].

Figure 2.10: Example objects

2.3 Neural Networks

2.3.1 Neurons

Brain is what allows to humans and animals to think ergo to learn, it controls every single step of all logical path that someone could generate. This complex system is pretty hard to understand however its primary components have a simpler form. Brain contains around 100 billion of neurons that are its basic building blocks, each one is typically connected to thousands of other neurons, they form a deep net of connections which is the tool used by the cerebrum to work. Let's have a look to fig.2.11 which illustrates their appearance and components.

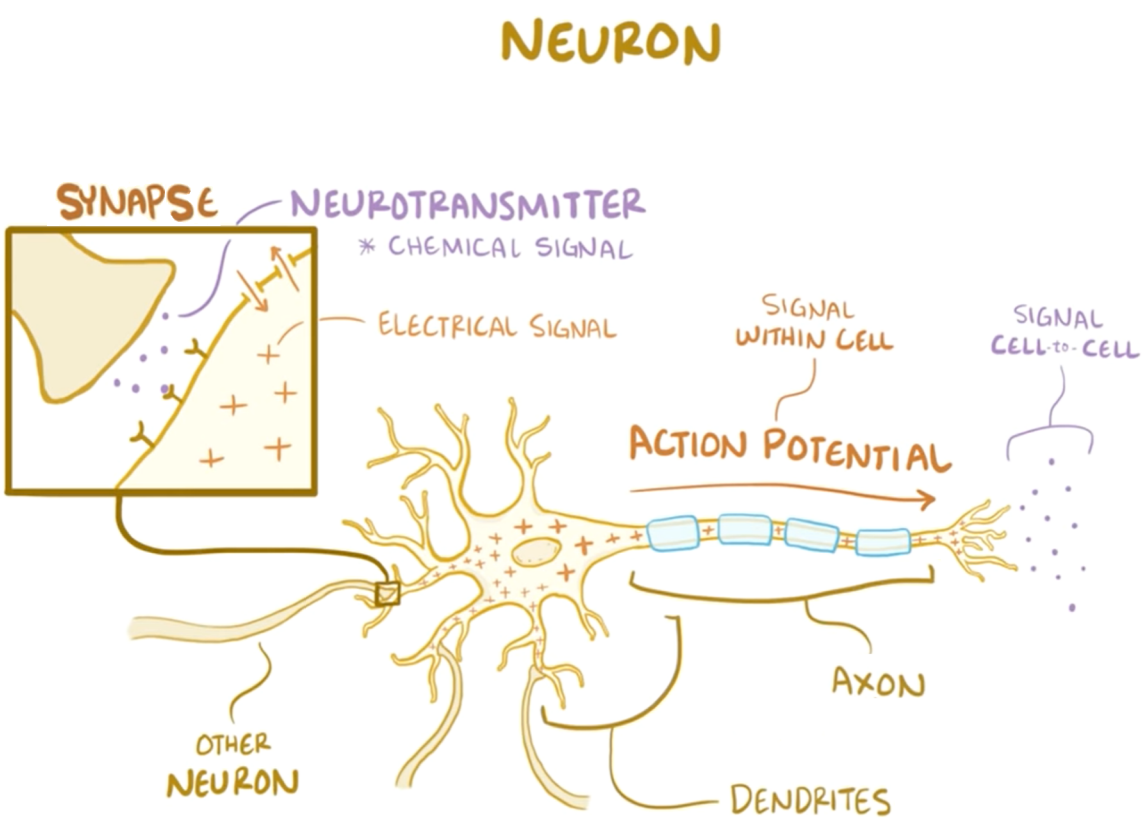


Figure 2.11: General Neuron Configuration

- **Axon:** The long, thin appendix in which action potentials are generated; it is the transmitting part of the neuron. After initiation, action potentials travel down axons to cause release of neurotransmitter into the synapse.
- **Dendrite:** The receiving part of the neuron. Dendrites receive synaptic inputs from axons, with the sum total of dendritic inputs determining whether the neuron will fire an action potential.
- **Action potential:** Brief (1 ms) electrical event typically generated in the axon that signals the neuron as 'active'. The action potential and consequent transmitter release allow the neuron to communicate with other neurons.
- **Neurotransmitter:** A chemical released from a neuron following an action potential. The neurotransmitter travels across the synapse to excite or inhibit the target neuron. Different types of neurons use different neurotransmitters and therefore have different effects on their targets.
- **Synapse:** The junction between the axon of one neuron and the dendrite of another, through which the two neurons communicate.

Neurons are essentially electrical devices, there are many channels sitting in the cell membrane that allow positive or negative ions to flow into and out of the cell. The cell's resting membrane potential is -70 mV. This membrane potential is not static, constantly going up and down, depending mostly on the inputs coming from the axons of other neurons. Some inputs increase the neuron's membrane potential, e.g. from -70 mV to -65 mV, and others do the opposite. These are respectively named excitatory and inhibitory inputs, as they promote or inhibit the generation of action potentials, the nature of a stimulus depends on the neurotransmitter released by the sending neuron.

Action potentials are the fundamental units of communication between neurons and they occur when the total sum of all of the excitatory and inhibitory inputs makes the neuron's membrane potential reach around -50 mV as in fig. 2.12, this value is called the action potential threshold. It is said that when a neuron exceeds this threshold it has "fire a spike" or "spiked". The term is a reference to the shape of an action potential as recorded using sensitive electrical equipment [20].

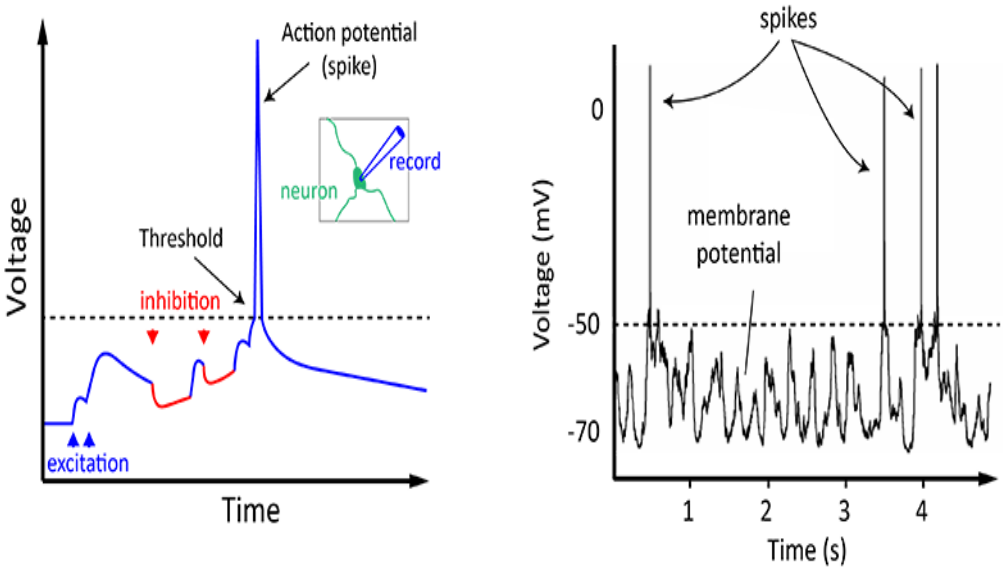


Figure 2.12: A neuron spikes when a combination of all the excitation and inhibition it receives, makes it reach threshold. On the right there is an example from an actual neuron in the mouse's cortex. Image: Alan Woodruff / QBI

Hebb's Rule

There is another interesting aspect to take in consideration, how do neurons activities modify their connection?
This is a concept described by Donald Hebb in 1949. Hebb says that "when the axon of a cell A is close enough to excite a B cell and takes part on its activation in a repetitive and persistent way, some type of growth process or metabolic change takes place in one or both cells, so that increases the efficiency of cell A in the activation of B".
It can be resumed as:

'neurons that fire together wire together'

The simultaneous activation of nearby neurons leads to an increase in the strength of synaptic connection between them. It is important to note that the neurons must be previously connected, sufficiently close to one another, so that the synapse can be reinforced [21].

Since Machine Learning wants to bring in the digital dimension the natural concept of learning, the Hebb's rule with all the similarity between neurons and electronic system, eases this digitalization implementation.

McCulloch-Pitts Neuron

The purpose of a mathematical model is to extract only the bare essentials required to accurately represent the entity being studied, removing all of the extraneous details. Warren McCulloch and Walter Pitts in the 1943 produced the first neuron model:

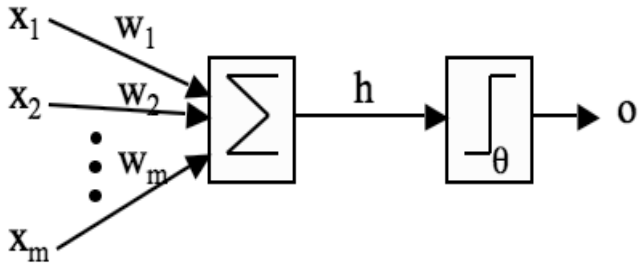


Figure 2.13: McCulloch and Pitts' mathematical model of a neuron.

There are three main components:

- a set of weights, w_i, that correspond to the synapses
- an adder that sums the input signals (equivalent to the membrane of the cell that collects electrical charge)
- an activation function (initially a threshold function) that decides whether the neuron fires ('spikes') for the current inputs

In the left part of fig. 2.13 there are a set of input nodes (labelled x_1, x_2, ... x_m,) with numerical values, in real neurons those inputs come from the outputs of other neurons. Each of these input values flows along a synapse characterized by a strength, called weight, to arrive at the neuron. The strength of the synapse affects the power of the signal, so the input is multiplied by the weight of the synapse (so $x_1 \cdot w_1$ and $x_2 \cdot w_2$, etc.). Now when all of these signals arrive into the neuron, it adds them up to see if the the value exceeds the threshold. It is represented by this formula:

$$h = \sum_{i=1}^{m} x_i \cdot w_i \tag{2.1}$$

which just means sum all the inputs multiplied by their synaptic weights.
The McCulloch and Pitts neuron is a binary threshold device, it sums up the inputs (multiplied by the synaptic strengths or weights) and either fires (produces output 1) or does not fire (produces output 0) depending on whether the input is above some threshold called θ.
We can write the second half of the work of the neuron, the decision about whether or not to fire (which is known as an activation function), as:

$$o = g(h) = \begin{cases} 1 & if \quad h > \theta \\ 0 & if \quad h < \theta \end{cases} \tag{2.2}$$

Let's consider an example with the following initial situation:

inputs: $x_1 = 1$, $x_2 = 1$, $x_3 = 0$
weights: $w_1 = 0.5$, $w_2 = 0$, $w_3 = -2$
threshold: 0.2

Using 2.1 the result is *g(h)* = 0.5 + 0 - 0 = 0.5; then applying 2.2, since 0.5 > 0.2, the neuron fires and the output is the logical 1.

This model might look simple, nevertheless a network of such neurons can perform any computation that a normal computer can, provided that the weights w_i are chosen correctly [11]. These neurons, or very simple variations, are commonly used with slightly different activation functions as ReLu and Sigmoid for most of neural networks:

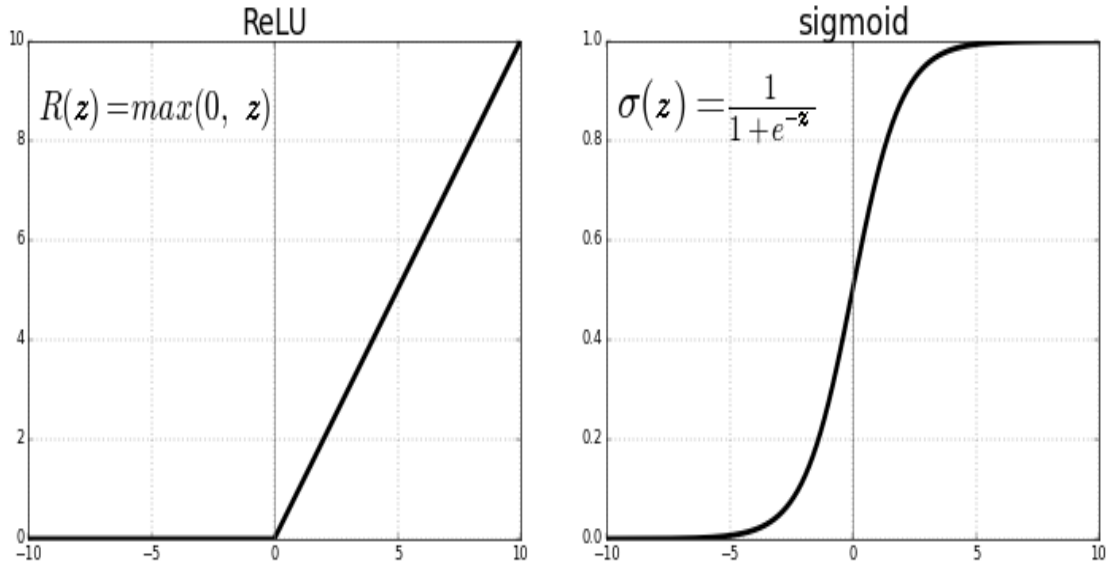


Figure 2.14: ReLU vs Sigmoid Function

The Sigmoid function, is used because it exists between (0 to 1), perfectly suitable for models which have to predict the probability as an output [22].
The ReLU (Rectified Linear Unit) Activation Function it is used in almost all the convolutional neural networks or deep learning. The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time. This means that the neurons will only be deactivated if the output of the linear transformation is less than 0. For the negative input values, the result is zero, that means the neuron does not get activated. Since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to the sigmoid [23].

2.3.2 Perceptron

Artificial neuron is an interesting tool but alone cannot do much, even connecting thousands of them together, things do not change because there is still a problem: they are unable to learn how they are. The learning is based on the changing so the question now is *"What can be modified to make them learn?"*
There are three values at stake: inputs, weights and threshold. Inputs are external so they cannot be modified, while weights and threshold are the keys of learning. This shows how the neural networks rely more on connections than on neurons. Weights represent the synapses and for a single element there are many synapses but a single threshold, this gives more freedom degrees to them respect to θ.

Now that it is know what to change, the next question is *"How?"*
The first answer to this question was the Perceptron. The Perceptron is a collection of McCulloch and Pitts neurons together with a set of inputs and some weights to fasten the inputs to the neurons.

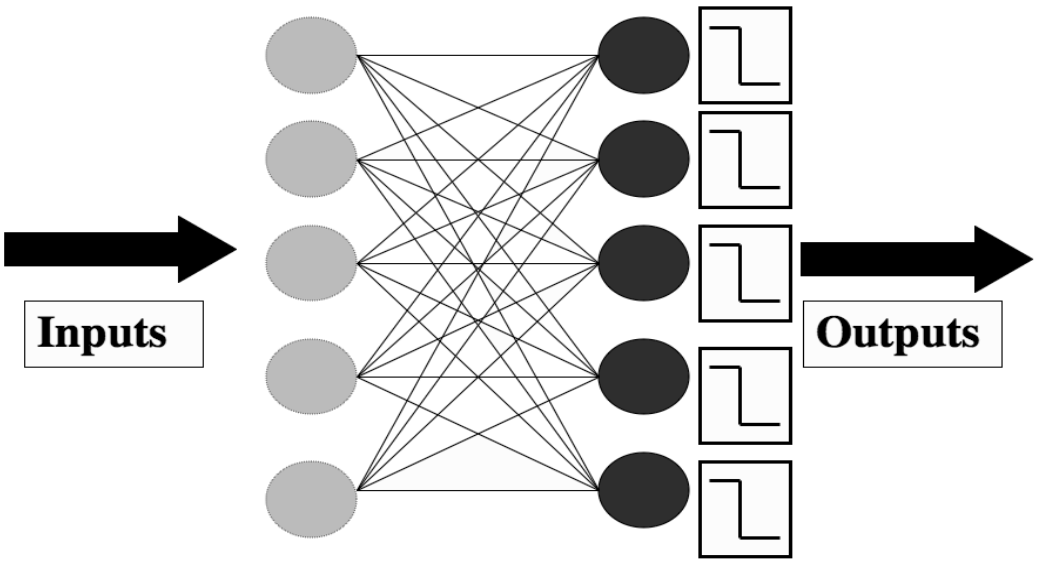


Figure 2.15: Perceptron Network

The network is shown in Fig. 2.15, inputs are the shaded light grey circles on the left. Neurons are on the right formed of two parts, the additive parts are drawn as black circles while the thresholder are the boxes next to them.

Neurons in the Perceptron are completely independent of each other: it does not matter to any neuron what the others are doing, it works out whether or not to fire by multiplying together its own weights and the input, adding them together, and comparing the result to its own threshold, regardless of other neurons are doing.

Even the weights which go into each neuron are separated for every one, so the only thing they share is the inputs, since every neuron sees all of them from the network.

In Fig. 2.15 the number of inputs is the same as the number of neurons, but this does not have to be the case — in general there will be m inputs and n neurons. The number of inputs is determined by the data and so is the number of outputs, in supervised learning, so the desire Perceptron behavior is to learn to reproduce a particular target, a pattern of firing and non-firing neurons for the given input. For McCulloch and Pitts neuron the weights are labelled as w_i, with the i index running over the number of inputs. Here, to work out which neuron the weight feeds into, they are labeled as w_{ij} , where the j index runs over the number of neurons. For instance w_{43} is the weight that connects input node 4 to neuron 3. An implementation of the neural network can use a two-dimensional array to hold these weights.

Working out whether or not a neuron should fire is easy, it is enough to set the values of the input nodes to match the elements of an input vector and then use equations 2.1 and 2.2 for each neuron. It is possible to do that for all neurons and the result is a pattern of firing and non-firing neurons, which looks like a vector of 0s and 1s; so if there are 5 neurons then a typical output pattern could be (1, 0, 0, 1, 0), which means that the first and fourth neurons fired and the others did not. The following step is comparing that pattern to the target, which is the known correct answer for this input, to identify which neurons got the answer right, and which did not. For a neuron that it is correct, nothing change; though any neuron that fired when it should not have done, or failed to fire when it should, its weights need to be fixed.

The main problem is to know how the weights should be, which is the point of the neural network, the goal is to change the weights in order to get the right answer the next time.

Let's suppose that an input vector occurs to the network and a neuron gets the wrong answer (its output does not match the target). There are m weights that are connected to that neuron, one for each of the input nodes. If the wrong neuron is labeled k, then the weights of interest are w_{ik}, i runs from 1 to m. At this point it is known what to change but it is still missing how.

The first thing to check is whether each weight is too big or too small computing $\Delta = y_k - t_k$ (the difference between the output y_k, which is what the neuron did, and the target for that neuron, t_k, which is what the neuron should have done). If it is negative the neuron should have fired and did not, the weights need to raise their values and vice versa if it is positive, this can be fixed subtracting the error value. However in case the input is negative, this adjustment switch the values over. If the neuron should fire, it is necessary to make the value of the weight negative as well.

To get around this, it is possible to multiply those two things together to see how it should change the weight:

$$\Delta w_{ik} = -(y_k - t_k) \cdot x_i \tag{2.3}$$

the new value of the weight will be the old value plus Δw_{ik}.

How much to change the weight is done by multiplying the value above by a parameter called the learning rate, usually labeled as η. This parameter decides how fast the network learns. The final rule for updating a weight w_{ij} is:

$$w_{ij} \leftarrow w_{ij} - \eta(y_j - t_j) \cdot x_i \tag{2.4}$$

The other thing to realise is that the network needs to see every training example multiple times. The first time the network might get some of the answers correct and some wrong; the next time it will hopefully improve, and eventually its performance will stop enhancing. Working out how long to train the network for is not easy, it is possible to define a maximum number of iterations "T". Of course, if the network got all of the inputs correct, then this would also be a good time to stop.

The *Learning Rate* η controls how much to change the weights by. A Perceptron with $\eta = 1$ makes the weights change a lot whenever there is a wrong answer, causing instabilities and it never settles down. The cost of having a small learning rate is more iterations before weights change significantly, the net takes longer to learn even if it will be more stable and resistant to data noise (errors).
The key is a trade-off with a moderate learning rate, typically $0.1 < \eta < 0.4$, depending upon how much errors are expected in the inputs.

As said before in order to make the neurons learn is possible to change weights and threshold. This θ should be adjustable to change the value that the neuron fires at. In case all of the inputs to a neuron are zero, it does not matter what the weights are (since zero times anything equals zero), the only way to control whether the neuron fires or not is through the threshold. If it was not adjustable and a neuron is supposed to fire when all the inputs to the net were zero, while another is supposed to not fire, it would be a problem. No matter what values of the weights were set, the two neurons would do the same thing since they had the same threshold and the inputs were null.
The trouble is that changing the threshold requires an extra parameter to write code for and this can be complex. Fortunately, there is a neat way around this problem, it consists in fixing the value of the threshold for the neuron at zero then adding an extra weight to the neuron which is connected to a fixed input value. That

weight is included in the update algorithm like all the other weights, so nothing new is necessary. Now the weights will change to make the neuron fire or not fire, whichever is correct, when an input of all zeros is given, since the extra input is always -1. This is called a **bias node** and its weights are usually given a 0 subscript, so that the weight connecting it to the j_{th} neuron is w_{0j} .

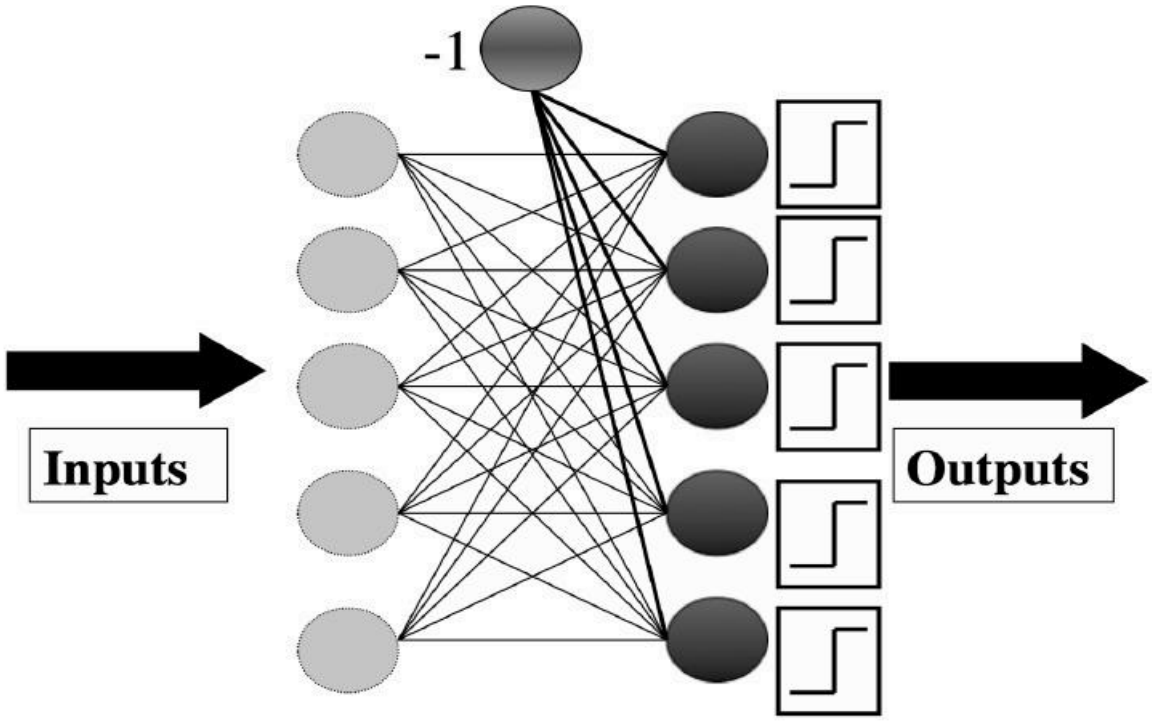


Figure 2.16: Perceptron Network with the bias input

The Perceptron Learning Algorithm is divided into two parts: a training phase and a recall phase.
The recall is used after training, it is supposed to be fast, since it will be used far more often than other phase. The training phase adopts the recall equation, since it has to work out the activations of the neurons before the error can be calculated and the weights trained [11].

Algorithm

- **Initialisation**
  - set all of the weights w_{ij} to small (positive and negative) random numbers
- **Training**
  - for T iterations or until all the outputs are correct:
    - for each input vector:
      - compute the activation of each neuron j using activation function g:

$$y_j = g\left(\sum_{i=0}^{m} w_{ij}x_i\right) = \begin{cases} 1 & \text{if } \sum_{i=0}^{m} w_{ij}x_i > 0 \\ 0 & \text{if } \sum_{i=0}^{m} w_{ij}x_i \leq 0 \end{cases}$$

      - update each of the weights individually using:

$$w_{ij} \leftarrow w_{ij} - \eta(y_j - t_j) \cdot x_i$$

- **Recall**
  - compute the activation of each neuron j using:

$$y_j = g\left(\sum_{i=0}^{m} w_{ij}x_i\right) = \begin{cases} 1 & \text{if } w_{ij}x_i > 0 \\ 0 & \text{if } w_{ij}x_i \leq 0 \end{cases}$$

2.3.3 Multi-Layer Perceptron

The Perceptron has a good potential but unfortunately it cannot be implemented for real world problems because of its linear nature which makes it unsuitable for complex issues, practical applications need an higher-level model.
As discussed before the learning is accomplished using the weights, more weights it means more computing power. There are two possibilities:

–add backwards connections, the output neurons connect to the inputs again, this is stated as Recurrent Networks.

–add neurons between the input nodes and the outputs

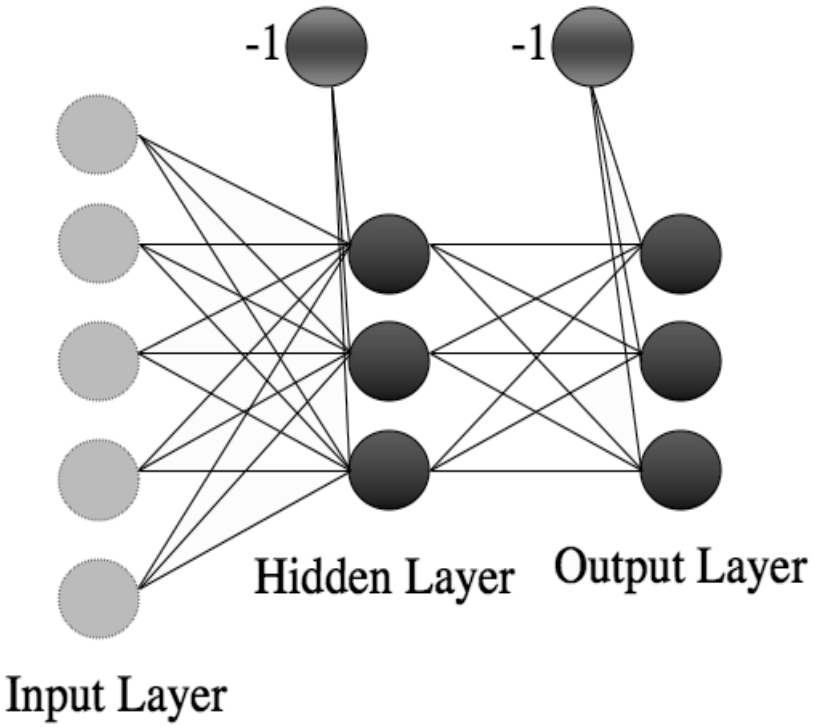


Figure 2.17: Multi Layer Perceptron Model

Let's focus on the second options. Now the problem to solve is how to train the network so that the weights are adapted to generate the correct (target) answers. The method used for the Perceptron needs to compute the error at the output and that's fine since the targets are known, but the weights which were wrong are unknown: those in the first layer, or the second? Worse, the correct activations for the neurons in the middle of the network are undefined. This fact gives the neurons in the middle of the network their name; they are called the hidden layer (or layers),

depicted in Fig. 2.17, because it is not possible to examine and correct their values directly.

However, a solution is the Multi-layer Perceptron (MLP), which is still one of the most commonly used Machine Learning methods around. MLP is a deep, artificial neural network composed of an input layer to receive the signal, an output layer which makes a decision or prediction about the input, and between those two, an arbitrary number of hidden layers that are the true computational engine of the MLP.
Backpropagation is used to make weights and bias adjustments relative to the error, and the error itself can be measured in a variety of ways, including by root mean squared error (RMSE).
Feedforward networks such as MLPs are like tennis, or ping pong, they are mainly involved in two motions, a constant back and forth:

- **forward pass:** the signal flow moves from the input layer through the hidden layers to the output layer, and the decision of the output layer is measured against the ground truth labels.
- **backward pass:** using backpropagation and the chain rule of calculus, partial derivatives of the error function w.r.t. the various weights and biases are back-propagated through the MLP. That act of differentiation produces a gradient along which the parameters may be adjusted to move the MLP one step closer to the error minimum. This can be done with any gradient-based optimisation algorithm such as stochastic gradient descent [24].

Multi-layer Perceptron uses complex models as backpropagation and chain rule of calculus that may be difficult to understand, a deeper description would shift the focus from the main topic, for this reason their explanations stops here but a mention to these mathematical methods was due.
However, for completeness, in the next page is reported the MLP algorithm.

Algorithm

- **Initialisation**
  - initialise all weights to small (positive and negative) random values
- **Training**
  - repeat:
    - for each input vector:

      Forwards phase:

      - compute the activation of each neuron j in the hidden layer(s) using:

$$h_\zeta = \sum_{i=0}^{L} x_i v_{i\zeta}$$

$$a_\zeta = g(h_\zeta) = \frac{1}{1+\exp(-\beta h_\zeta)}$$

      - work through the network until you get to the output layer neurons, which have activations:

$$h_\kappa = \sum_{j} a_j w_{j\kappa}$$

$$y_\kappa = g(h_\kappa) = \frac{1}{1+\exp(-\beta h_\kappa)}$$

      Backwards phase:

      - compute the error at the output using:

$$\delta_o(\kappa) = (y_\kappa - t_\kappa)\, y_\kappa (1 - y_\kappa)$$

      - compute the error in the hidden layer(s) using:

$$\delta_h(\zeta) = a_\zeta (1 - a_\zeta) \sum_{k=1}^{N} w_\zeta \delta_o(k)$$

      - update the output layer weights using:

$$w_{\zeta\kappa} \leftarrow w_{\zeta\kappa} - \eta \delta_o(\kappa) a_\zeta^{\text{hidden}}$$

      - update the hidden layer weights using:

$$v_\iota \leftarrow v_\iota - \eta \delta_h(\kappa) x_\iota$$

    - (if using sequential updating) randomise the order of the input vectors so that you don't train in exactly the same order each iteration
  - until learning stops
- **Recall**
  - use the Forwards phase in the training section above

2.4 CNN & RNN

2.4.1 Convolutional Neural Network

The CNNs are a family of Neural network widely employed in computer vision, usually with data that have a spatial relation.
They can be represented in a graph with a series of steps, from the starting complex input are extracted a series of informations, called features, gradually simpler and more recognizable for the computer.
CNN is a levels architecture, typical non-cyclical, where the most important levels are the convolutional layers after which is named

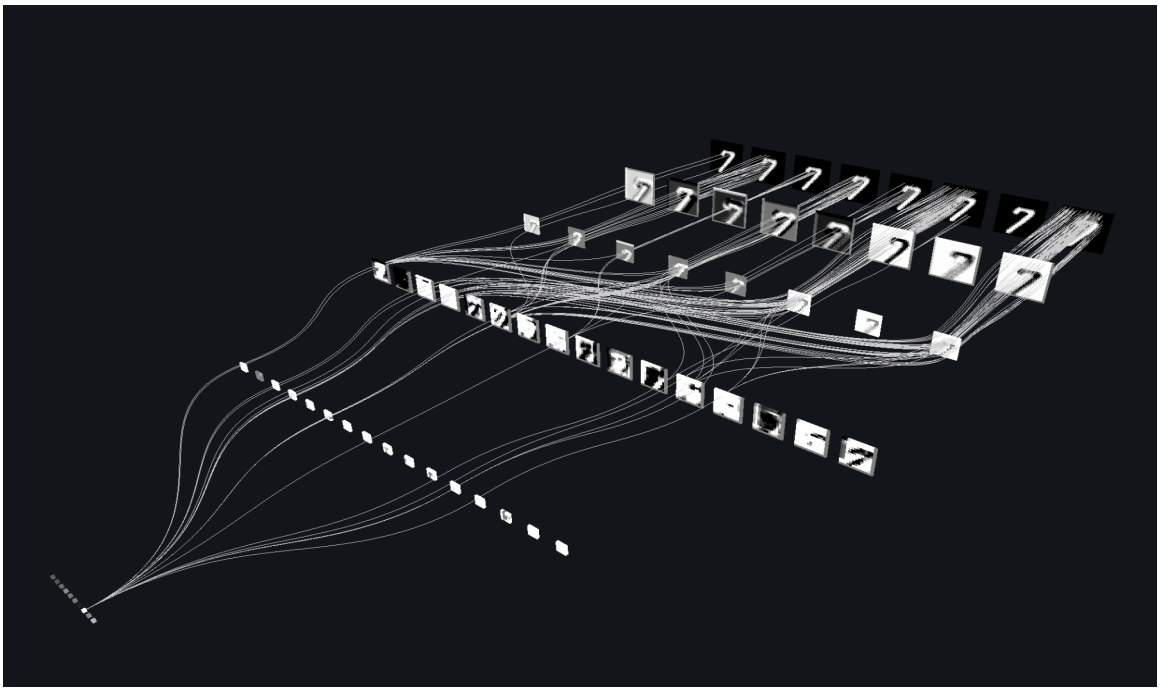


Figure 2.18: Building Network Commands

Convolution is a mathematical operation where the result is the translation of one function on the other as the time changes. Since typical CNNs receive bidimensional input data, like images for example, they get one or more matrices and one called **kernel** that will be translated to compute its result, called **feature map**. The translation step is called **stride**, it represents how much to move inside the matrix.

The size of the Kernel and its stride are set at the beginning as hyperparameters. Fig. 2.20 shows an example step by step with a 3x3 Kernel, with a stride equal to 1, passing over an image represented with a 5x5 matrix, the resulting features are stored in a 2x2 matrix.

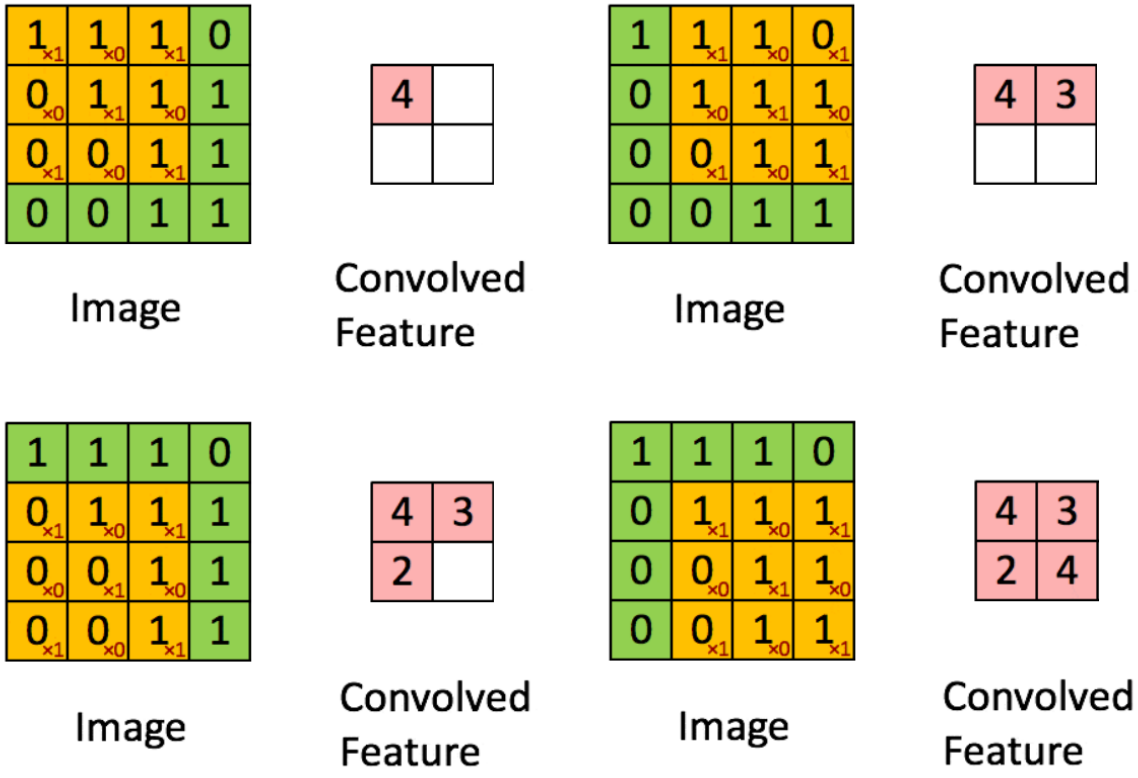


Figure 2.19: Convolutional 3x3 Kernel with a stride of 1

After a convolutional layer it is common to find a subsampling which reduce the dimension of the matrix. The most used subsampling techniques are:

- **MaxPooling:** every cell of the resulting matrix stores the maximum value of the staring matrix.

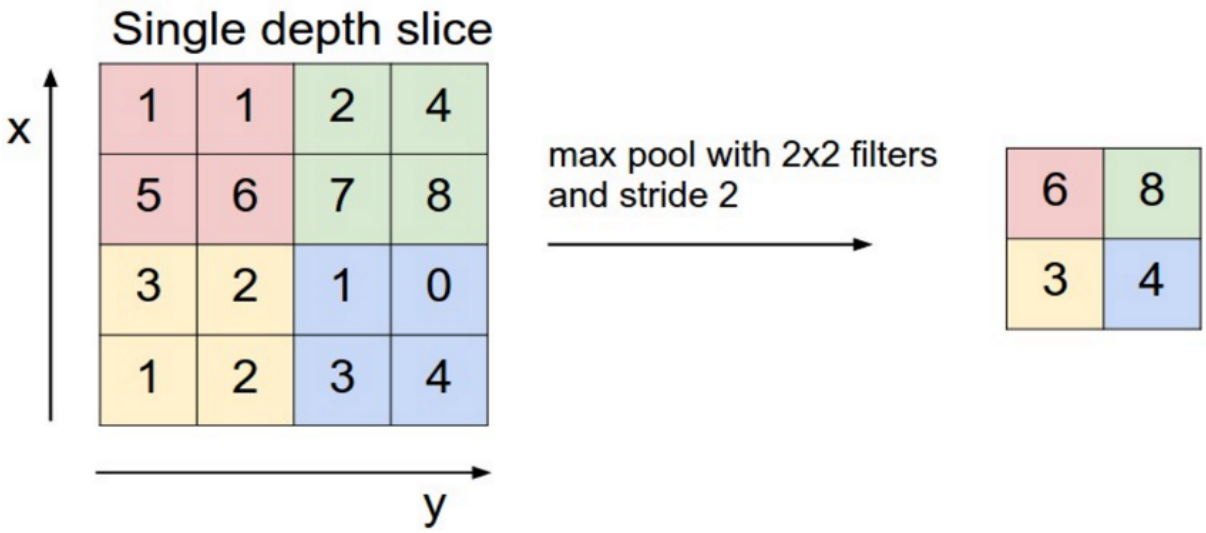


Figure 2.20: MaxPooling example

- **AveragePooling:** every cell of the resulting matrix stores the average value of the staring matrix.

CNNs generally use a fully connected layers too, which can be seen as neuron vectors. The output of each neuron is transmitted to all the neurons of the following layers with different weights [25].

2.4.2 Recurrent Neural Network

A Recurrent Neural Network (RNN) is a net which processes sequences (as daily stock prices, sentences, or sensor measurements) one element at a time while retaining a memory, called a state, of what has come previously in the sequence. Differently from CNN a RNN has not a fixed length for the inputs and outputs.

Recurrent means the output at the current time step becomes the input to the next time step. At each element of the sequence, the model considers not just the current input, but what it remembers about the preceding elements.

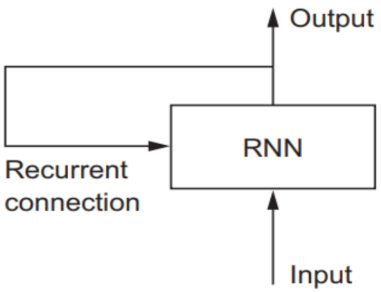


Figure 2.21: Recurrent loop

This memory allows the network to learn long-term dependencies in a sequence which means it can take the entire context into account when making a prediction, whether that be the next word in a sentence, a sentiment classification, or the next temperature measurement. A RNN is designed to mimic the human way of processing sequences: humans consider the entire sentence when forming a response instead of words by themselves. For example, consider the following sentence: "The concert was boring for the first 15 minutes while the band warmed up but then was terribly exciting."
A machine learning model that considers the words in isolation would probably conclude this sentence is negative. An RNN by contrast should be able to see the words "but" and "terribly exciting" and realize that the sentence turns from negative to positive because it has looked at the entire sequence. Reading a whole sequence gives us a context for processing its meaning, a concept encoded in recurrent neural networks [26].

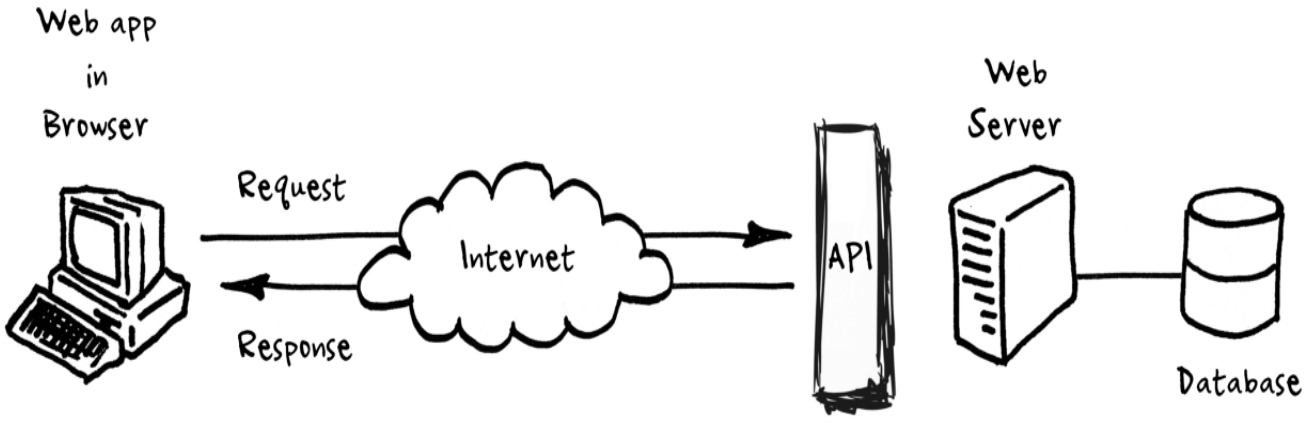
2.5 Software

So far the all the fundamentals necessary to reach the goal of this project have been discussed, there is still a missing particular to decide before to start working: "Where is better to develop the network?"
In the previous sections it is explained what is the final goal, what is the Machine Learning and how it works, though there is no reference to a real development environment, that's the aim of this section.
Machine Learning is being so common that there are many solutions to implement a network, in this case only two of them has been taken in consideration: Keras and Matlab.

2.5.1 Keras

Application Programming Interface (API)

In basic terms, APIs just allow applications to communicate with one another. When people speak of "an API", they sometimes generalize and actually mean "a publicly available web-based API that returns data, likely in JSON or XML". The API is not the database or even the server, it is the code that governs the access point(s) for the server [27]. It may be seen like a set of definitions and protocols that determines the communications with server facilitating the informations exchange.



Keras & TensorFlow

Keras is a deep learning API written in Python, running on top of the machine learning platform TensorFlow. It was developed with a focus on enabling fast experimentation, being able to go from idea to result as fast as possible is the key to do good research.
TensorFlow is an end-to-end, open-source machine learning platform and Keras is its high-level API: an approachable, highly-productive interface for solving machine learning problems, with a focus on modern deep learning. It provides essential abstractions and building blocks for developing and shipping machine learning solutions with high iteration velocity[28].

Keras is a great way to start with Neural Networks because it allows to easily upload pretrained nets and make adjustments in order to achieve better performance using the fine-tuning technique. Moreover it is written in Python a user friendly code rich of intuitive commands which is pretty simple to begin with, even for a novice programmer..

Practical Example

Python has a wide range of implementations, therefore there are several applications examples. A good starting point to know Keras in the image recognition field is the "Cats & Dogs" algorithm, which simple program able to distinguish cat and dog in pictures, it is awesome how it works with so few data, only 2000 training examples (1000 per class), and yet can even recognize the breeds of dogs and cats. There are three version, let's see the basic one for a better description.

In order to make the most of the few training examples, they will be "augmented" via a number of random transformations, so that the model would never see twice the exact same picture. This helps prevent overfitting and helps the model generalize better.
In Keras this can be done via the **keras.preprocessing.image.ImageDataGenerator** class. This class allows to:

- configure random transformations and normalization operations to be done on your image data during training
- instantiate generators of augmented image batches (and their labels) via *.flow*(data, labels) or *.flow_from_directory*(directory).

Let's look at the code fragment illustrated in Fig. 2.22, used as example, where a picture of a couple of cats is used to generate more samples, Fig. 2.23, using this tool and save them to a temporary directory (rescaling in this case it is disabled to keep the images displayable):

```
from keras.preprocessing.image import ImageDataGenerator, array_to_img, img_to_array, load_img

datagen = ImageDataGenerator(
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')

img = load_img('data/train/cats/cat.0.jpg')  # this is a PIL image
x = img_to_array(img)  # this is a Numpy array with shape (3, 150, 150)
x = x.reshape((1,) + x.shape)  # this is a Numpy array with shape (1, 3, 150, 150)

# the .flow() command below generates batches of randomly transformed images
# and saves the results to the `preview/` directory
i = 0
for batch in datagen.flow(x, batch_size=1,
                          save_to_dir='preview', save_prefix='cat', save_format='jpeg'):
    i += 1
    if i > 20:
        break  # otherwise the generator would loop indefinitely
```

Figure 2.22: Code Snippet of Data Augmentation

There are much more options available, the ones reported in the figure above have different roles:

- *rotation_range* is a value in degrees (0-180), a range within which to randomly rotate pictures
- *width_shift* and *height_shift* are ranges (as a fraction of total width or height) within which to randomly translate pictures vertically or horizontally
- *rescale* is a value which multiply the data before any other processing. Original images consist in RGB coefficients in the 0-255, but such values would be too high for these models to process (given a typical learning rate), instead values between 0 and 1 become the target by scaling with a 1/255 factor

- *shear_range* is for randomly applying shearing transformations [1]
- *zoom_range* is for randomly zooming inside pictures
- *horizontal_flip* is for randomly flipping half of the images horizontally –relevant when there are no assumptions of horizontal asymmetry (e.g. real-world pictures).
- *fill_mode* is the strategy used for filling in newly created pixels, which can appear after a rotation or a width/height shift.

Figure 2.23: Contents of the directory

The right tool for an image classification job is a convnet (CNN). Since there are only few examples, number one concern should be overfitting.
Data augmentation is one way to fight overfitting, but it isn't enough since augmented samples are still highly correlated. The main focus for fighting overfitting should be the entropic capacity of model (the maximum quantity of informations the model is allowed to store). A model that can store a lot of information has the potential to be more accurate by leveraging more features, but it is also more at risk to start storing irrelevant features. Meanwhile, a model that can only store a few features will have to focus on the most significant features found in the data, and these are more likely to be truly relevant and to generalize better.

[1] A transformation in which all points along a given line L remain fixed while other points are shifted parallel to L by a distance proportional to their perpendicular distance from L. Shearing a plane figure does not change its area [29].

There are different ways to modulate entropic capacity. The main one is the choice of the number of model hyperparameters, i.e. the number of layers and the size of each layer.
In this case it is used a very small convnet with few layers and few filters per layer, alongside data augmentation and dropout[2]. Dropout also helps reduce overfitting, by preventing a layer from seeing twice the exact same pattern, thus acting in a way analogous to data augmentation; it could be said that both dropout and data augmentation tend to disrupt random correlations occuring in your data.

The code snippet in Fig. 2.24 reports the command to built this net, a simple stack of 3 convolution layers with a ReLU activation and followed by max-pooling[3] layers, on top are placed two fully-connected layers. The model ends with a single unit and a sigmoid activation, which is perfect for a binary classification.

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense

model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(3, 150, 150)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# the model so far outputs 3D feature maps (height, width, features)

model.add(Flatten())  # this converts our 3D feature maps to 1D feature vectors
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

Figure 2.24: Building Network Commands

[2]Dropout is a technique used to prevent a model from overfitting. Dropout works by randomly setting the outgoing edges of hidden units (neurons that make up hidden layers) to 0 at each update of the training phase [30].

[3]Maximum pooling, or max pooling, is a pooling operation that calculates the maximum, or largest, value in each patch of each feature map. This highlight the most present feature in the patch, not the average presence of the feature as in the case of average pooling [31].

This approach gets a validation accuracy of 79-81% after 50 epochs [32].
It's a surprisingly result considering that all the training code it's just about 40 lines, moreover even better performance have been reached by the other two versions.
This is further confirmation of how much Keras is user friendly, that's explains why it was considered for this project.

2.5.2 Matlab

The second proposal to achieve the goal imposed is Matlab, which is a language and interactive environment for developing algorithms, performing numerical computation, analyzing and visualizing data.

The Matlab language supports the vector and matrix operations that are fundamental to engineering and scientific problems. Commands can be executed one at a time, providing immediate results. It is possible to create scripts and functions to reuse and automate work. Matlab provides the features of a traditional programming language, as well as layout tools for designing custom graphical user interfaces. Add-on toolboxes extend the environment to solve problems in a range of applications, including signal processing and communications, video and image processing, control design and Machine Learning. Matlab code and results can be automatically published. Algorithms and applications can be distributed as standalone executables, components for integration and other software environments, such as Excel or as portable C code for algorithms using a subset of the language [33].

This is a summary in a few words, it really offers a broad variety of instruments to develop applications, however there is a drawback.
The programming language is not intuitive and user friendly as Python, even making small adjustments on a working code it requires time. Also the structure is more complex, simple project folder can be divided in a series of file paths containing working functions, setup files, data folders and so on. A proper management of this hierarchical framework requires some efforts, for this reason creating a Neural Network with Matlab is not recommended for beginners.

However, despite the higher difficulties, this project starts from a training Matlab code [8], so the smoothest way to proceed is keep going on this road since converting the initial algorithm in Python could require a lot of time resulting a useless effort, considering that Matlab could still achieve better results.

Development

The target of this labour is a working CRNN (i.e. the combination of pretrained Convolutional Neural Network with a Recursive Neural Network) able to recognize objects from images. The union between these two different kind of approach is supposed to significantly enhance the performances because it takes the best of both.
In this chapter it will be discussed how this network has been made but before to start it is important to have a solid knowledge of its components. The aim of this part is to explain the single parts and their roles arriving to a description of the final architecture and its implementation.

3.1 CRNN for Object recognition

3.1.1 VGG-f

The CNN chosen is the VGG-f, it is an eight layer deep net. Its structure is reported on Fig. 3.1, which has been originally designed and trained for image classification. The input image size is 224x224x3. Fast processing is ensured by the four pixel stride in the first convolutional layer and it has been trained on ILSVRC. Data augmentation in the form of horizontal, flips, random crops, and RGB color jittering has been applied in the learning process.

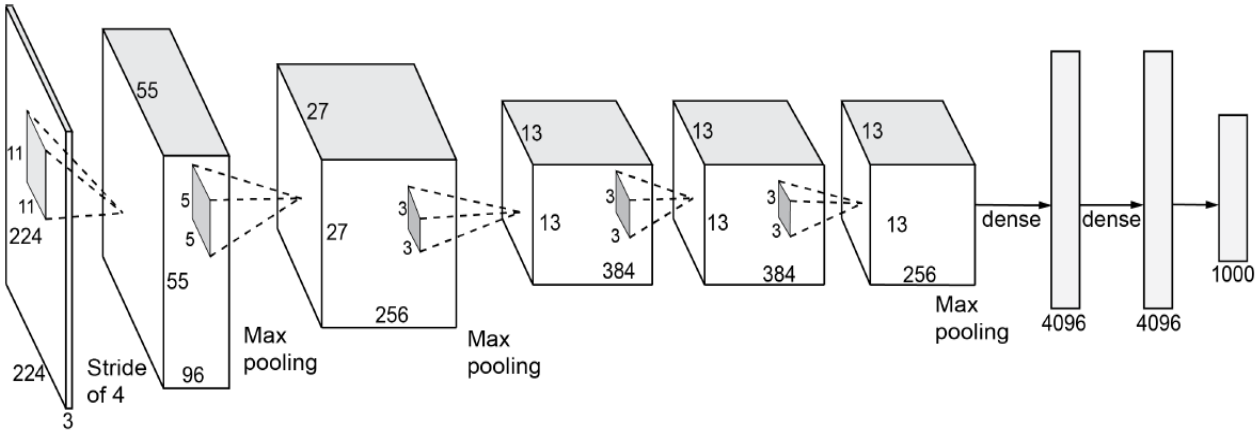


Figure 3.1: Architecture of VGG-f

The deep features can be extracted after removing the last classification layer consisting of 1000 neurons. The only image pre-processing is to resize the input images to the network input size and to subtract the average image, which is provided together with the network parameters [34].

3.1.2 Recursive Neural Network

The RNN implemented in this CRNN it is a Recursive Neural Network, it is just a generalization of a recurrent network, where the weights are shared (and dimension remains constant) along the length of the sequence, because it is impossible to deal with position-dependent weights when it encounters a sequence at test-time of different length of the one shown at train-time.
In a recursive network the weights are shared at every node for the same reason. This means that all the w_{xh} weights will be equal and so will be the w_{hh} weight, that is true because it is a single neuron which has been unfolded in time.
How a Recursive Neural Network looks like it is shown in Fig. 3.2.

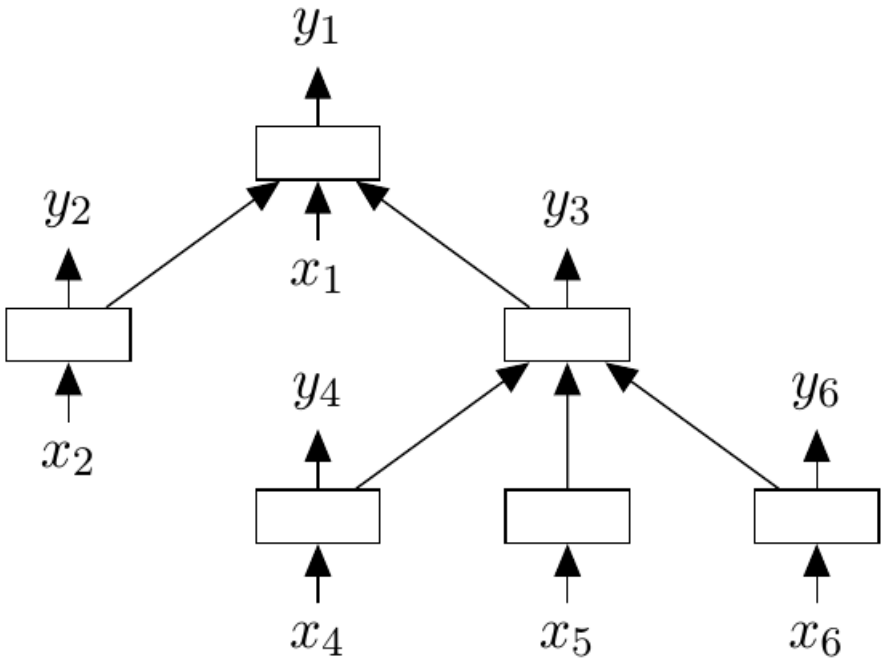


Figure 3.2: Recursive tree

This RNN has a tree-like structure, each children node are simply a reproduction to the parent node. It creates hierarchical representations for tasks needing a parse tree organization [35].

3.1.3 CRNN

This paper is based on the work of Hieu Minh Bui and his collaborators in their treatise *"Object Recognition Using Deep Convolutional Features Transformed by a Recursive Network Structure"*, the contribution of their approach to the computer vision can be summarized in two points:

- A new efficient feature extraction method using a deep convolutional network structure trained on a large dataset for object recognition tasks represented by a smaller dataset. The method combines the well known AlexNet with a RNN structure.
- Re-evaluating the use of AlexNet as a feature extractor to determine applicability of the 'best layer'.

A number of studies have confirmed that intermediate layers in a deep network can capture features that provide a good trade-off between representation and object

independence. In that work, several low-level layers of the AlexNet trained on the ImageNet data set were selected, and each of these layers were examined as a black-box feature extractor [8].
The only difference applied in this project consists in using the VGG-f model instead of AlexNet as pretrained CNN, which is still trained using fixed size RGB images from ImageNet.
The VGG-f structure, reported in Fig. 3.1 and in the top of Fig. 3.3, consists of 8 layers, where the first 5 layers are convolutional and the remaining 3 layers are fully connected. The last fully-connected layer has the form of a Softmax classifier [1] to categorize an input image into one of the classes used in training.

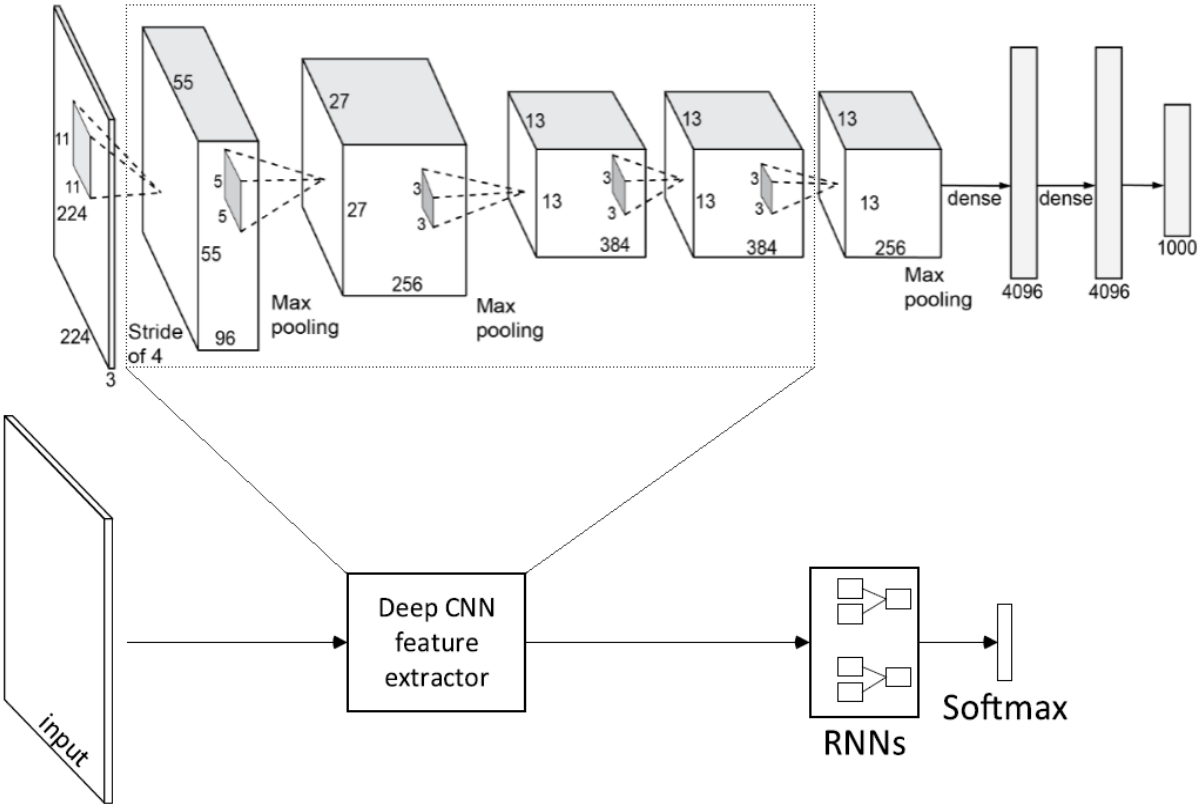


Figure 3.3: Structure of the proposed system

The proposed recognition structure, with part of VGG-f embedded as the feature

[1]The Softmax classifier gets its name from the softmax function, which is used to squash the raw class scores into normalized positive values that sum to one, so that the cross-entropy loss can be applied. Softmax classifier provides probabilities for each possible class, the highest value corresponds to the final prediction [36]

extractor, is illustrated in Fig. 3.3. This new structure uses the same input format as the VGG-f and consists of several low-level layers of it.

The RNN can be implemented in different versions depending on the number of input units, this choice has a considerably influence on performance. A high value of this parameter allows to reach a better accuracy with the drawback of longer computational time, meanwhile a low number makes the net faster but more imprecise.
Once this feature is set it takes care of the extracted features, before feeding them into the Softmax classifier that performs recognition on the target data set.

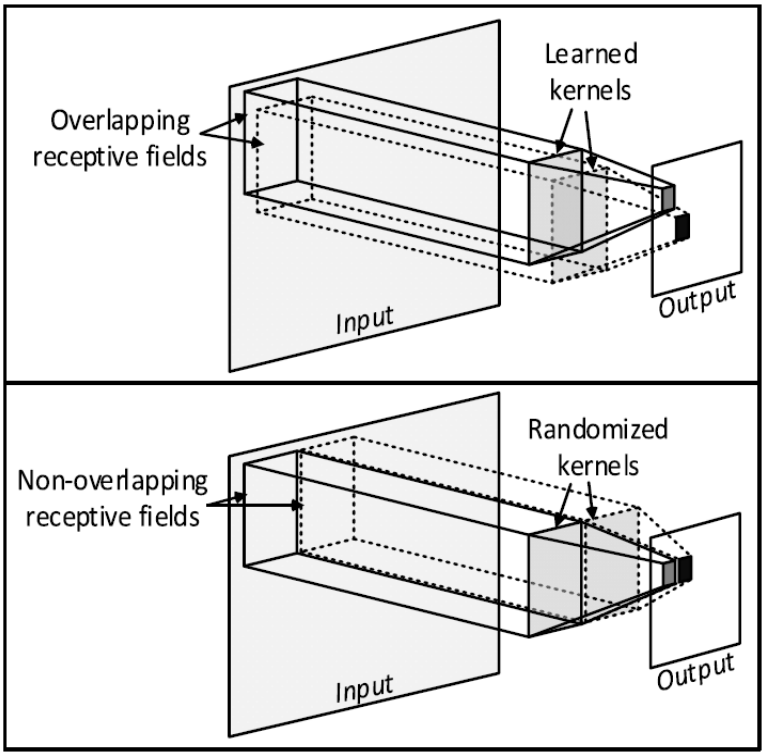


Figure 3.4: CNN layer (top) and RNN layer (bottom).

As illustrated in Fig. 3.4, the RNN layer is quite similar to the structure of the CNN. Both structures divide the input data into patches of equal size, compute element-wise products of each patch with a shared array of weights, add the outcomes together and then process the sum through a sigmoidal or other type of squashing function.
The RNN is different to the CNN in two aspects. Firstly, the RNN set of weights is randomly initialized based on the input data structure, while the CNN set of weights is learned from the data.
Secondly, the RNN uses non-overlapping input patches while the CNN typically uses densely overlapped patches. Due to the non-overlapping attributes of patches,

the RNN is computationally less expensive than the CNN. The recursive structure allows RNNs to capture repetitive patterns in the input, while the random weights and nonlinear squash function help to differentiate between class clusters.

The fully connected layer number 6 of the VGG-f provided the highest quality features for the object classification task. In a fully-connected layer, each neuron connects in the same way to all neurons of the previous layer, and as a result the spatial information that describes the input data is largely discarded. The current idea is to explore the possibility of utilizing the remaining spatial characteristics of data processed by the deep network through application of the RNN structure. Therefore, it is necessary to examine the features produced by previous CNN layers, where spatial information still remains.
Consequently, this work analyzes and compares the efficiency of applying RNN processing, with different sizes, to CNN features with regards to an object recognition task. Features produced by a deep neural network layer, either CNN or fully-connected, typically have the form of a 3-dimensional matrix of size $w \cdot h \cdot d$.
The 3rd, 4th, and 5th CNN layers of VGG-f set $w = h = 13$, whilst the value of d was set to either 384 for the 3rd and 4th layers and to 256 for the 5th.
In this case the layer used to extract features is the 5th, it passes them to the assembly of RNNs, the output matrix is set to ensure compatibility with the RNNs. The RNN outputs were then concatenated to form the final representation of the original input image. The output features from the RNNs were used to train a classifier, the Softmax, powered by the Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) algorithm, it was applied to make predictions about what object contains the input image minimizing the cross entropy error [37].

3.2 Training & Propagation

3.2.1 Training

According to what as been said so far the two main parts of the structure have not to be trained, because the CNN is already trained and the RNN have random initialized weights that do not have to be modified.
However just putting together those two networks is not sufficient, there is still a component that needs to understand how to recognize objects; it is the Softmax Classifier. This is the last element of the system, its role is to take the final decision about the prediction. It receives the output of the RNN and through some compu-

tations gets a vector of the same size of the all possible object classes. Each value of the vector assigns a probability to the object corresponding to its index, the index assigned to the highest value matches the most probable object.

The Classifier in its calculations uses a parameter called θ, it is obtained as result of the processing of the Washington RGB samples handled by trained networks.
To shed lights about training it is better analyze the code provided by Hieu Minh Bui and coworkers, there are many functions forming this algorithm, the most relevant parts are mentioned below while the entire code is reported in appendix B.1.

Code functions

- **runCRNN:** It is called by the main function *"Launcher"* which just starts the program. It sets the library paths and recalls others function as *initParams, forwardCNN* and *test_numRNN*. It stores the data structures returned by *forwardCNN* in external matlab files, in such a way that if it is necessary they can be directly uploaded from those files in every desired program.

```
function [combineAcc rgbAcc depthAcc] = runCRNN()
startTime = tic;
addpath(genpath('../vlFeat/matconvnet-1.0-beta20'));
addpath(genpath('../minFunc'));
params = initParams();
disp(params);
%% Run RGB
disp('Forward propagating RGB data');
params.depth = false;
[cnnTrain cnnTest] = forwardCNN(params);
save('cnnTrain.mat','cnnTrain', '-v7.3');
save('cnnTest.mat','cnnTest', '-v7.3');
test_numRNN
disp('Elapsed time: ');
toc(startTime)
return;
```

Figure 3.5: runCRNN

- **initParams:** It sets some hyperparameters and other features necessary to work. It sets the path to the dataset, the model of the CNN and its number of filters, which CNN layer to connect to the input of the RNN and the size of the latter.

```
function params = initParams()
params.recogDisp = 1;
params.dataFolder = '../W_RGBD_dataset/data';
params.split = 2;
params.numFilters = 256;
params.numRNN = 128;
run('vl_setupnn.m')
params.cnnModel.net = load('../vlFeat/matconvnet-1.0-beta20/matlab/imagenet-vgg-f.mat');
params.layer = 13;
```

Figure 3.6: runCRNN

- **forwardCNN:** This is the function that takes care of the features extraction. Following the path specified as the variable *params.dataFolder* in the *initParams* function, images from the dataset are collected and passed to the CNN network. Using several functions those images are split between Train and Test, then are re-elaborated and at the end two data structures are returned:

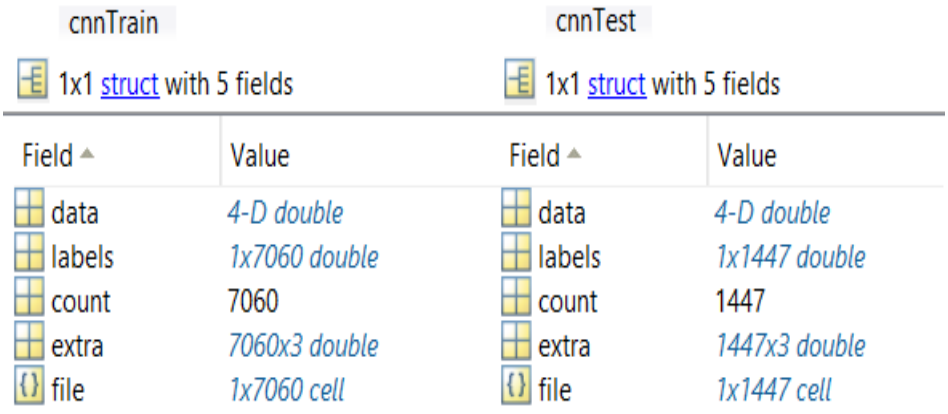


Figure 3.7: Features

In the Fig 3.7 is reported a view of the results, where “data” is a 4 dimension

matrix of size 7060x256x13x13 for the Train and 1447x256x13x13 for the Test: the first size is the number of images, 256 is the number of filters and the two dimension of 13 are due to the CNN layers measures. Then there is a "label" for each input sample and other values negligible in order to understand how it works.

- **test_numRNN:** This is the part of the code which uses all the available variations of RNN. As first thing it clears the memory and sets again the parameters that it needs, in this way it can be run independently from the first half of the code, in fact it invokes again the *initParams* function and uploads the features previously extracted and stored in the external files ("cnnTrain.mat" and "cnnTest.mat").

```
clear
addpath(genpath('../vlFeat/matconvnet-1.0-beta20'));
addpath(genpath('../minFunc'));
params = initParams();
params.RFS = [13 13];
numRNN_array = [128, 64, 32, 16, 8, 4, 2, 1];
acc = zeros(10,length(numRNN_array));
count = 1;
for i = numRNN_array
    load('cnnTest.mat');load('cnnTrain.mat');
    params.numRNN = i;
    disp(['numRNN = ' num2str(params.numRNN)]);   )
    rnn{1} = initRandomRNNWeights(params, size(cnnTest.data,2));
    save('rnn_64', 'rnn', '-v7.3');
    [rnnTrain, rnnTest] = forwardRNN(cnnTrain, cnnTest, params, rnn);
    save('RNN_output', 'rnnTrain', 'rnnTest', '-v7.3'
    for spl = 1:10
        params.split = spl;
        disp(['SPLIT: ' num2str(spl)]);
        [rnnTrain, rnnTest] = split_config(rnnTrain, rnnTest, spl);
        disp(['train_count: ' num2str(rnnTrain.count) '   test_count: ' num2str(rnnTest.count)]);
        [rgbAcc, y_predict, confidence] = trainSoftmax(rnnTrain, rnnTest, params);
        acc(spl, count) = rgbAcc;
    end
    disp(['Num of RNN: ' num2str(params.numRNN)]);
    disp(['Average accuracy = ' num2str(mean(acc(:,count))) ' +/- ' num2str(std(acc(:,count)))]);
    count = count + 1;
end
```

Figure 3.8: test_numRNN

It is based on two nested *for-cycle*, the external one is used to try all the different sizes of network stored in the *numRNN_array* while the internal is used to iterate between 10 different data splits. For each size the weights of the RNN are randomly generated then the *forwardRNN* function is called, it takes the extracted features and the weights and generates two new data structures, as it was done for the CNN.

Those data structures are subsequently split according to the iteration of the internal cycle and then employed as arguments for the *trainSoftmax*, which returns the performance obtained for each combination of *numRNN_array* value and data organization.
It is still missing a code snippet not shown in Fig. 3.8, but it has just the purpose of computing the average of the results achieved for every RNN dimension and displaying them on the monitor, this allows the user to compare the possible solutions and to asses the best one depending on its needs.

- **forwardRNN:** When it is called it receives as argument the features extracted by the CNN, through some computation cycles it returns two new data structures almost identical to the ones in input with only a difference, the voice *data* is now containing a 2-Dimension matrix where the first value is the product of the number of CNN filters multiplied by the RNN size and the second one is the number of images for that set.

```
function [train test] = forwardRNN(train, test, params, rnn)
train.data = forward(train.data, rnn, params);
test.data = forward(test.data, rnn, params);
train.data = train.data(:,:)';
test.data = test.data(:,:)';
function rnnData = forward(data, rnn, params)
data = permute(data,[2 3 4 1]);
[numMaps, rows, cols, numImgs] = size(data);
RFS = params.RFS;
numRNN = params.numRNN;
assert(rows == cols);
depth = floor(log(rows)/log(RFS(1)) + 0.5);
assert(mod(log(rows)/log(params.RFS(1)),1) < 1e-15);
assert(mod(log(cols)/log(params.RFS(2)),1) < 1e-15);
rnnData = zeros(numRNN, numMaps, numImgs);
for r = 1:numRNN
 W = squeeze(rnn{1}.W(r,:,:));
 tree = data;
 for layer = 1:depth
     newTree = zeros(numMaps,size(tree,2)/RFS(1),size(tree,3)/RFS(2),numImgs);
     rc = 1;
     for row = 1:RFS(1):size(tree,2)
         cc = 1;
         for col = 1:RFS(2):size(tree,3)
             newTree(:,rc,cc,:) = tanh(W * reshape(tree(:,row:row+RFS(1)-1,col:col+RFS(2)-1,:),[],numImgs));
             cc = cc + 1;
         end
         rc = rc + 1;
     end
     tree = newTree;
 end
 rnnData(r,:,:) = squeeze(tree);
end
rnnData = permute(rnnData, [3 1 2]);
```

Figure 3.9: forwardRNN

- **trainSoftmax:** It is the piece of code dedicated to train the final layer, the Classifier. By means of functions exploiting mathematical processes it generates a variable *"Wcat"*, as illustrated in Fig. 3.10, which is the fundamental component in the final formula to get the prediction. This formula is applied in the function *softmaxTest* but it will be deepen in Section 3.2.2.

```
function [percentCorrect, y_predict, confidence] = trainSoftmax(train, test, params)
addpath('../minFunc');
options.maxIter = 350;
k = max(train.labels);
n = size(train.data,1);
Wcat = 0.005 * randn(k,n);
options.Method = 'lbfgs';
options.display = 'on';
lambda = 1e-8;
[X decodeInfo] = param2stack(Wcat);
X = minFunc(@softmaxCost, X, options, train.data, train.labels, lambda,decodeInfo, params);
Wcat = stack2param(X,decodeInfo);
[percentCorrect, y_predict, confidence] = softmaxTest(Wcat,test, params);
disp([datestr(now)  ' percent correct: ' num2str(percentCorrect)]);
return

function [train test] = addExtraFeatures(train, test)
extraStd = 12;
m = mean(train.extra);
s = std(train.extra)/extraStd;
train.extra = bsxfun(@rdivide,bsxfun(@minus,train.extra,m),s);
test.extra = bsxfun(@rdivide,bsxfun(@minus,test.extra,m),s);

train.data = [train.data;train.extra'];
test.data = [test.data;test.extra'];
return
```

Figure 3.10: trainSoftmax

This was in broad outline the training, running this code is quite expensive in terms of time. The features extraction performed by the CNN does not require too much time considering the size of the Washington RGB, obviously a larger dataset requires longer time interval. The real problem in terms of time is the RNN, because a single execution is relatively short but multiple iterations due to the nested *for-cycle* significantly extends the completion.

SUMMARY:

Num_of_RNN	Accuracy%	Std_dev%
128	84.786	2.3534
64	83.913	2.6369
32	83.818	2.2178
16	82.521	2.4654
8	80.012	1.608
4	75.483	2.1013
2	65.782	1.3561
1	50.878	2.098

```
Start   Time: 13-May-2020 20:03:20
End     Time: 14-May-2020 00:24:15
Elapsed Time: 04:20:55
```

Figure 3.11: Summary Report

In the figure above is reported the log file containing the final statistics.
The results show that the program execution has lasted for more than four hours, this interval refers to a situation where eight different sizes have been tested for ten times, this means eighty iterations. Changing the number of cases or the resources employed to run the code means a variation in the elapsed time.

3.2.2 Propagation

In order to utilize the desired CRNN network there are two important variables to save in external files after the training: the *"RNN_weights"* and *"Wcat"*.
The former is necessary to set the right values on the RNN neurons while the latter is used in the formula to make predictions. All other data produced during the training are unnecessary since the CNN was pre-trained.
Those two external files are then uploaded in a new program with the same structure of the training one but with some small differences, the functions are similar but they are now storing a lighter code modified to work with a single image at time. The complete algorithm is reported in appendix B.2.

Code functions

- **propagation:** It is the main function and it consists of few commands. It recalls other function as *initParams*, *forwardCNN*, *forwardRNN* and *softmaxTest* then it prints on the display the name of the predicted object.

```
addpath(genpath('../matconvnet-1.0-beta20'));
addpath(genpath('../minFunc'));
startTime = tic;
params = initParams();
CNN_features = forwardCNN(params);

% load("RNN_weights");

RNN_output = forwardRNN(CNN_features, params, rnn);

% load('Wcat');

[maximum, y_predict] = softmaxTest(Wcat, RNN_output);
load("classes");
disp('prediction')
disp(classes(y_predict))
toc(startTime)
```

Figure 3.12: propagation

In Fig. 3.12 is reported the code lines, it can be noticed that there are two lines commented, this because they are needed just once to upload in the working environment the two matlab files generated during the training.

When Matlab has stored those data it uses them as arguments for *forwardRNN* and *softmaxTest*.

- **forwardCNN:** This version is a bit easier with respect to the one in the training code, that's because now it has to treat just one image. It takes the target file from a folder called *"input"* then passes it to the net and return its features in the form of a 3-Dimension matrix of size 256x13x13.

```
function [CNN features] = forwardCNN(params)
instanceData = dir([params.dataFolder '/*.jpg']);
disp(instanceData)
fileImgName = [params.dataFolder  instanceData(1).name];
img = imread(fileImgName);
img = imresize(img, [224, 224]);
img = single(img);
fim = extract_Alexnet(img, params.cnnModel.net, params.layer);
CNN_features = permute(fim, [3, 2, 1]);
return

function fileBool = isValid(name)
fileBool = (~strcmp(name,'.') && ~strcmp(name,'..') && ~strcmp(name,'.DS_Store'));
return;

function data = cutData(data)
data.data = data.data(1:data.count,:,:,:);
data.extra = data.extra(1:data.count,:);
return
```

Figure 3.13: forwardCNN for propagation

- **forwardRNN:** As for the *forwardCNN* this is a simpler version of the one for the training. It takes the features extracted by the CNN and returns a single vector with a number of elements equal to the product of CNN filters multiplied by the RNN size.

```
function [RNN_output] = forwardRNN(CNN_features, params, rnn)
disp('Forward Prop RNN...');
RNN_output = forward(CNN_features, rnn, params);
RNN_output = RNN_output(:,:)';

function rnnData = forward(data, rnn, params)
data = permute(data,[1 2 3]);
[numMaps, rows, cols] = size(data);
numImgs=1;
RFS = params.RFS;
numRNN = params.numRNN;
assert(rows == cols);
assert(mod(log(rows)/log(params.RFS(1)),1) < 1e-15);
assert(mod(log(cols)/log(params.RFS(2)),1) < 1e-15);
rnnData = zeros(numRNN, numMaps, numImgs);
for r = 1:numRNN
    W = squeeze(rnn{1}.W(r,:,:));
    tree = tanh(W * reshape(data,[],1));
    rnnData(r,:) = squeeze(tree);
end
rnnData = permute(rnnData, [3 1 2]);
```

Figure 3.14: forwardRNN for propagation

- **softmaxTest:** It takes *"Wcat"* as argument under the name of *theta* while the other argument is the vector generated by the RNN. After a brief computation it returns the maximum value associated to an object and its index, which will be used to select the right answer between all the possible classes.

```
function [maximum, y_predict] = softmaxTest(theta,x)
    num = exp(theta*x);
    value = num/sum(num);
    [maximum, y_predict] = max(value);
end
```

Figure 3.15: softmaxTest

Finally it can be said that, as the files obtained with the training are uploaded in the Matlab environment, it is possible to get a prediction every time is wanted just calling the function *propagation*.
This makes the program useful for every device with a camera, when the image captured by the camera is passed to the program it returns its evaluation after a response time depending on the complexity of the RNN.

Once the system recognizes what it is facing, it can regulate itself by changing its behavior or continuing to work as before.

3.3 Results

Since this program can be employed with different types of RNN it can achieve different results in terms of accuracy and time to response. In Fig. 3.11 are shown even the accuracy performances of each RNN size that has been tested.
As it could be expected the larger nets give more precise predictions, but there is not such a sharp distinction between the first five versions, in fact the first and the fifth have a rate discrepancy lower than the 5%, therefore they reach almost the same results.
The real difference is in term of velocity, this produce a huge gap in reaction speed and this is a critical issue in case of a real time application.
Let's consider an empirical experiment to clarify the divergences among the nets.
Giving as input the same image to identical structures which differ only in the RNN sizes, the prediction and time may change. In Fig.3.16 is illustrated an input image took from the web, not present in the Washington RGB, with the networks outcomes.

RNN:	Elapsed time	Prediction
128	27.035282 seconds	'banana'
64	14.686872 seconds	'banana'
32	7.781696 seconds	'banana'
16	4.138225 seconds	'banana'
8	2.646080 seconds	'banana'
4	2.328017 seconds	'banana'
2	2.106477 seconds	'toothbrush'
1	2.053499 seconds	'potato'

Figure 3.16: RNN performances compared on a banana image [3]

The first thing that jumps out is the huge amount of time required for the 128-RNN, while decreasing the size considerably decrease the elapsed time, getting in almost all the cases the correct prediction. There are only two nets which produce a wrong guess and thinking on their accuracy it was predictable (2-RNN has an accuracy rate about 65.782% and 1-RNN 50.878%).
It is interesting to note how time is almost halved by exponentially reducing the size of the structure, this improvement saturates with the 8-RNN. In fact, the reduction in the number of input units no longer generates appreciable results except for a worsening in reliability.

Limits

This algorithm has obtained satisfactory achievements in recognizing objects but it has some limitations. It is true that it recognizes with a remarkable success percentage the items present in its training set, but it is also true that the variety of objects is not so large, in fact it contains only 51 classes.
However there is a more serious problem, the code has learned from a series of cropped images in order to focus only in the typical aspects of the target. This makes it more sensible to the noisy informations coming from the background. For example the code has no difficulties to recognize a lemon in the two images below.

Figure 3.17: Input examples of lemon images [4][5]

The problems emerge when the input image contains disturbing elements like in Fig. 3.18, it is still containing a lemon but it has a greater resolution than the one the program is set for, so it is not able to isolate the single object from the green leaves around, concluding its evaluation with a wrong result: "*greens*".
This underlines the necessity of the program to make some modifications before starting the analysis.

Figure 3.18: Noisy background example [6]

Conclusions

The efforts done thus far result in a working code able to recognize objects belonging to one of the training classes. It comes in multiple versions of itself and depending on the specifics of the task one can be more suitable than another.
For a real time application the 128-RNN can be embedded in a system as long as it works with a sufficiently powerful hardware, otherwise it will turn out useless because its slow respond. But similar resources often take up large areas making the entire device unwieldy.
On the other hand, a 1-RNN is pretty fast, it has an excellent answering velocity, the instrument on which this code is loaded could be very small and light with the added value of a return almost instantaneous; but there is a crucial issue with this version, the accuracy. This model is so imprecise to be less reliable than the full VGG-f alone, this makes it unsuitable for nearly all the applications.
In this case where the desired product is a code able to work with a camera inserted on a drone, it is necessary a trade-off among the two previously cited models, which are the extremes regarding efficiency and speed.

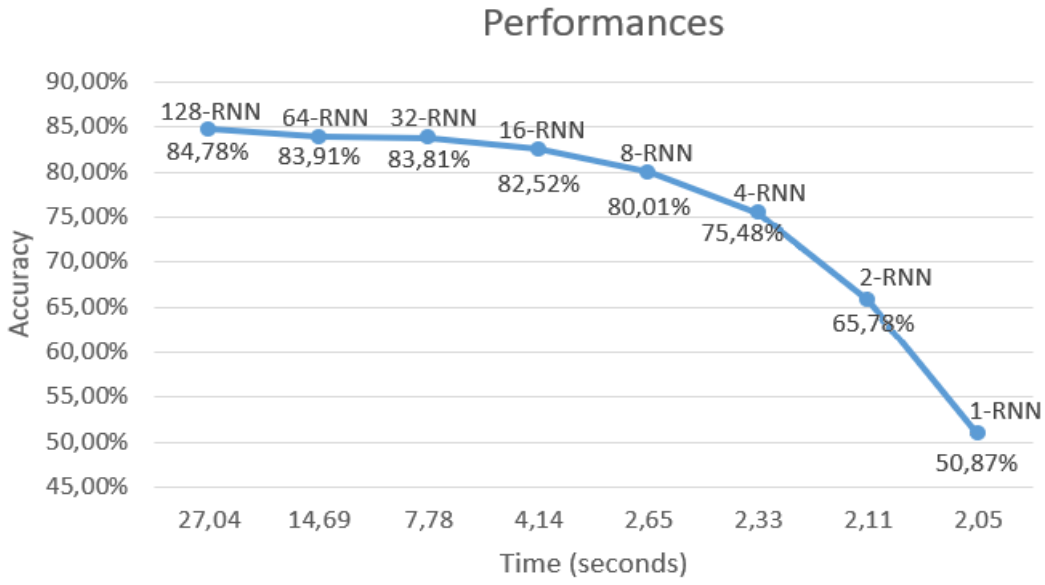


Figure 4.1: Performances over the banana image input graphically represented

The 128-RNN requires a bulky tools not appropriate for a tiny flying device like a drone, while the 1-RNN has too severe performance gaps.
The most eligible candidates are the versions in the middle, but there is one which represents the best solution for this Project: 8-RNN.
This is the most balanced configuration, as said before a further reduction in size, 4-RNN, produces a minimal improvement in speed while there is a clear worsening in terms of precision, around 5%. Despite this code is sufficiently light to be upload in a small device it reaches rather good performance, those features make the 8-RNN the most appropriate model for the purpose of this paper.

It is still important to remind that this code has some necessities for the pre-processing on the input image. When the input image is picked from its folder by the *forwardCNN* function, it is cropped in a matrix of size 224x224, this is a not negligible constraint for the camera. For example, if the picture has a resolution of 1 Megapixel (1280x960) this means that it contains more informations than what the code can elaborate so it ends to loss what could be crucial to recognize the object.
A possible solution can be a simple preprocessing step, identical to what has been seen in the "Cats & Dogs" code in section 2.5.1, where it was demonstrated how images have been pre-processed before to feed the algorithm, one of these treatment was the zooming. In fact zooming a part of the image, for example the center, will ensure that the input size of the data fits the dimensions the system is able to work with.
Using directly a camera with the same specifics could be a restriction for other works since nowadays 224x224 is a poor value for a camera, while using a better camera and perform a zoom can offer more flexibility, it is just needed that the target is positioned in the area of zooming.
Providing just some little changes the code will be able to be embedded in any working system for a real world applications without any problems.

Appendices

Appendix A

Cats & Dogs

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense
from keras import backend as K


# dimensions of our images.
img_width, img_height = 150, 150

train_data_dir = 'data/train'
validation_data_dir = 'data/validation'
nb_train_samples = 2000
nb_validation_samples = 800
epochs = 50
batch_size = 16

if K.image_data_format() == 'channels_first':
    input_shape = (3, img_width, img_height)
else:
    input_shape = (img_width, img_height, 3)

model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=input_shape))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

```
model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

# this is the augmentation configuration we will use for training
train_datagen = ImageDataGenerator(
    rescale=1. / 255,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True)

# this is the augmentation configuration we will use for testing:
# only rescaling
test_datagen = ImageDataGenerator(rescale=1. / 255)

train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
    validation_data_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    class_mode='binary')

model.fit_generator(
    train_generator,
    steps_per_epoch=nb_train_samples // batch_size,
    epochs=epochs,
    validation_data=validation_generator,
    validation_steps=nb_validation_samples // batch_size)

model.save_weights('first_try.h5')
```

Appendix B

B.1 Training Code

Launcher:

```
if(strcmp('on', get(0,'Diary')))
    diary off
end
diaryname = ['Log_' strrep(strrepdatestr(now),' ','_'),':','-')'.txt'];
diary(diaryname)
t = timer('timerfcn',@updatediary,'period',10,'ExecutionMode','fixedRate');
start(t)
%put program to run here
runCRNN

stop(t)
diary off
```

runCRNN:

```
function [combineAcc rgbAcc depthAcc] = runCRNN()
startTime = tic;
% init params
addpath(genpath('../vlFeat/matconvnet-1.0-beta20'));
addpath(genpath('../minFunc'));

params = initParams();
disp(params);

%% Run RGB

disp('Forward propagating RGB data');
params.depth = false;

[cnnTrain cnnTest] = forwardCNN(params);
```

```
save('cnnTrain.mat','cnnTrain', '-v7.3');
save('cnnTest.mat','cnnTest', '-v7.3');

test_numRNN

disp('Elapsed time: ');
toc(startTime)
return;
```

initParams:

```
function params = initParams()
params.recogDisp = 1; %value other than 0 will display all misclassified objects

% Set the data folder here
params.dataFolder = '../W_RGB_dataset/data';

% set the data split to train and test on
params.split = 2;

% set the number of CNN filters
params.numFilters = 256; %number of filters in selected layer of CNN model

% set the number of RNN to use
params.numRNN = 64;

%load the pretrained deep net model
run('vl_setupnn.m')
params.cnnModel.net = load('../vlFeat/matconvnet-1.0-beta20/matlab/
imagenet-vgg-f.mat');

%choose the layer in deep net to extract feature
params.layer = 13;
```

forwardCNN:

```
function [train test] = forwardCNN(params)
%% init the data containers
nf = params.numFilters;
% hard code final size
fi = 13; %set based on layer size of CNN model
fj = 13;

numExtra = 3;
train = struct('data',zeros(100,nf,fi,fj),'labels',[],'count',0,'extra,
zeros(100,numExtra),'file',[]);
```

```
test = struct('data',zeros(100,nf,fi,fj),'labels',[],'count',0,'extra',
zeros(100,numExtra),'file',[]);

%% load the split information
% testInstances specifies which instance from each class will used for testing
testInstances = splits(:,params.split);

% grab all categories in data folder
data = [params.dataFolder '/wrgb-dataset_eval'];
categories = dir(data);
numCategories = length(categories);

catNum = 0;
for catInd = 1:numCategories
    if mod(catInd,10) == 0
        disp(['---Category: ' num2str(catInd) ' out of '
        num2str(numCategories) '---']);
    end
    if isValid(categories(catInd).name)
        catNum = catNum+1;
        % grab all instances within this category
        instance = dir(fileCatName);
            if isValid(instance(instInd).name)
                % check if a testing instance then take data
                fileInstName = [fileCatName '/' instance(instInd).name ];

                if testInstances(catNum) == str2double(instance(instInd).name(regexp
                (instance(instInd).name,'[0-9]')))
                    test = addInstance(fileInstName, catNum, test, params);
                else
                    train = addInstance(fileInstName, catNum, train, params);
                end
            end
        end
    end
end

train = cutData(train);
test = cutData(test);
return

function fileBool = isValid(name)
fileBool = (~strcmp(name,'.') && ~strcmp(name,'..') && ~strcmp(name,'.DS_Store'));
return;

function data = addInstance(fileInstName, catNum, data, params)
```

```
searchStr = '/*_crop.png';

% grab image names from this instance
instanceData = getValidInds(dir([fileInstName searchStr]), fileInstName);

% set the labels
% data.labels at the beginning is empty
data.labels = [data.labels ones(1,length(subSampleInds))*catNum];

for imgInd = subSampleInds
    % data.count at the beginning is 0
    data.count = data.count + 1;

    % read in our file from disk
    fileImgName = [fileInstName '/' instanceData(imgInd).name];

    img = imread(fileImgName);
    img = imresize(img, [224, 224]);
    img = single(img);

    startInd = max(strfind(instanceData(imgInd).name,'_'));

    % extract deep features
    fim = extract_Alexnet(img, params.cnnModel.net, params.layer);
    fim = permute(fim, [3, 2, 1]); %from 13x13x256 to 256x13x13

    % add these features to data
    if data.count > size(data.data,1)
        data.data(end*2,end,end) = 0;
        data.extra(end*2,end) = 0;
    end
    data.data(data.count,:) = fim(:);

    % add for sanity check
    data.file{end+1} = instanceData(imgInd).name(1:startInd);
end
return


function data = cutData(data)
assert(length(data.labels) == data.count);
data.data = data.data(1:data.count,:,:,:);
data.extra = data.extra(1:data.count,:);
return
```

test_numRNN:

```
clear
addpath(genpath('../vlFeat/matconvnet-1.0-beta20'));
addpath(genpath('../minFunc'));

params = initParams();
% hard code the receptive field size (or child size) of each RNN
params.RFS = [13 13]; %set base on size of extracted CNN features

numRNN_array = [128, 64, 32, 16, 8, 4, 2, 1];

acc = zeros(10,length(numRNN_array));
count = 1;
for i = numRNN_array
    load('cnnTest.mat')
    load('cnnTrain.mat')
    params.numRNN = i;
    disp(['numRNN = ' num2str(params.numRNN)]);

    for r = 1:(floor(log(size(cnnTest.data,2))/log(params.RFS(1))+0.5) - 1)
        rnn = initRandomRNNWeights(params, size(rgbTest.data,2));
    end

    save('rnn_1', 'rnn', '-v7.3');

    %%using RNN
    [rnnTrain, rnnTest] = forwardRNN(cnnTrain, cnnTest, params, rnn);
     if (params.numRNN == 1)
        save('RNN_output', 'rnnTrain', 'rnnTest', '-v7.3');
    end
    clear cnnTrain cnnTest  %release some memory

    for spl = 1
        params.split = spl;
        disp(['SPLIT: ' num2str(spl)]);
        [rnnTrain, rnnTest] = split_config(rnnTrain, rnnTest, spl);
        disp(['train_count: ' num2str(rnnTrain.count) '
        test_count: ' num2str(rnnTest.count)]);
        [rgbAcc, y_predict, confidence] = trainSoftmax(rnnTrain, rnnTest, params);
        acc(spl, count) = rgbAcc;
    end
    disp(['Num of RNN: ' num2str(params.numRNN)]);
    disp(['Average accuracy = ' num2str(mean(acc(:,count))) ' +/- '
    num2str(std(acc(:,count)))]);

    count = count + 1; %tracking RNN profile
end
```

```
Accuracy = mean(acc);
Std_dev = std(a);
Num_of_RNN = numRNN_array;
t = table(Num_of_RNN', Accuracy', Std_dev','VariableNames',{'Num_of_RNN' 'Accuracy' '
disp 'SUMMARY:'
disp(t);
```

forwardRNN:

```
function [train test] = forwardRNN(train, test, params, rnn)
% In this function we take the final output from the CNN and stack an RNN
% ontop of the final responses. The children will be pooled spatially
% accross all maps.

% forward prop training data
disp('Forward Prop Train...');
% output : numTrain x numRNN x numHid
train.data = forward(train.data, rnn, params);
% forward prop testing data
disp('Forward Prop Test...');
test.data = forward(test.data, rnn, params);

train.data = train.data(:,:)';
test.data = test.data(:,:)';


function rnnData = forward(data, rnn, params)
data = permute(data,[2 3 4 1]);
[numMaps, rows, cols, numImgs] = size(data);

RFS = params.RFS;
numRNN = params.numRNN;
assert(rows == cols);
depth = floor(log(rows)/log(RFS(1)) + 0.5);

% ensure a balanced tree is possible with these sizes
assert(mod(log(rows)/log(params.RFS(1)),1) < 1e-15);
assert(mod(log(cols)/log(params.RFS(2)),1) < 1e-15);

rnnData = zeros(numRNN, numMaps, numImgs);
for r = 1:numRNN
    if mod(r,8)==0
        disp(['RNN: ' num2str(r)]);
    end
     W = squeeze(rnn{1}.W(r,:,:));
```

```
    tree = data;
    for layer = 1:depth
% W = squeeze(rnn{layer}.W(r,:,:)); % activate to use different
                                                % weights for each RNN layer
        newTree = zeros(numMaps,size(tree,2)/RFS(1),size(tree,3)/RFS(2),numImgs);
        rc = 1;
        for row = 1:RFS(1):size(tree,2)
            cc = 1;
            for col = 1:RFS(2):size(tree,3)
                newTree(:,rc,cc,:) = tanh(W * reshape(tree(:,row:row+RFS(1)-1,
                col:col+RFS(2)-1,:),[],numImgs));
                cc = cc + 1;
            end
            rc = rc + 1;
        end
        tree = newTree;
    end
    rnnData(r,:,:) = squeeze(tree);
end

rnnData = permute(rnnData, [3 1 2]);
```

trainSoftmax:

```
function [percentCorrect, y_predict, confidence] = trainSoftmax(train, test, params)
addpath('../minFunc');
options.maxIter = 350;
k = max(train.labels);
n = size(train.data,1);
Wcat = 0.005 * randn(k,n);
options.Method = 'lbfgs';
options.display = 'on';
lambda = 1e-8;
[X decodeInfo] = param2stack(Wcat);
X = minFunc(@softmaxCost, X, options, train.data, train.labels, lambda,decodeInfo, pa
Wcat = stack2param(X,decodeInfo);
save('Wcat_1', 'Wcat', '-v7.3');
[percentCorrect, y_predict, confidence] = softmaxTest(Wcat,test, params);
disp([datestr(now)  ' percent correct: ' num2str(percentCorrect)]);
return

function [train test] = addExtraFeatures(train, test)
extraStd = 12;
m = mean(train.extra);
s = std(train.extra)/extraStd;
train.extra = bsxfun(@rdivide,bsxfun(@minus,train.extra,m),s);
```

```
test.extra = bsxfun(@rdivide,bsxfun(@minus,test.extra,m),s);

train.data = [train.data;train.extra'];
test.data = [test.data;test.extra'];
return
```

param2stack:

```
function [stack decodeInfo] = param2stack(varargin)

stack = [];

for i=1:length(varargin)
    if iscell(varargin{i})
        for c = 1:length(varargin{i})
            decodeCell{c} = size(varargin{i}{c});
            stack = [stack ; varargin{i}{c}(:)];
        end
        decodeInfo{i} = decodeCell;
    else
        decodeInfo{i} = size(varargin{i});
        stack = [stack ; varargin{i}(:)];
    end
end
```

softmaxTest:

```
function [percentCorrect, y_predict, confidence] = softmaxTest(theta,test, params)
%Hieu changed: x, y -> test, then:
x = test.data;
y = test.labels;

correct = 0;
total = 0;
y_predict = zeros(size(y));
confidence = zeros(size(y));
raw_predict = zeros(max(y),length(y)); %Hieu added
label_cmpr = {'PREDICTED', 'REAL ITEM'}; %Misclassified list
for i = 1:length(y)
    num = exp(theta*x(:,i));
    raw_predict(:,i) = num;
    norm_predict = num/sum(num);
    [confidence(i), y_predict(i)] = max(norm_predict);
    %for comparing numbered labels
    if (y(i) == y_predict(i))
        correct = correct + 1;
```

```
    else %Hieu added
        tmp = find(y == y_predict(i),1);
        label_cmpr{i-correct+1,1} = test.file{tmp}(1:(regexp(test.file{tmp},'[0-9]') -
        label_cmpr{i-correct+1,2} = test.file{i};
    end
    total = total + 1;
end
if (params.recogDisp)
    disp(label_cmpr);
end

%save result (unnormalized) for later combination (if need)
file_str = sprintf('raw_predict_split%d_%dRNN.mat', params.split, params.numRNN);
time = datestr(now);
real_labels = test.labels;
save(file_str, 'raw_predict', 'real_labels', 'time', '-v7.3');
percentCorrect = (correct/total)*100;
end
```

B.2 Propagation Code

Launcher:

```
 if(strcmp('on', get(0,'Diary')))
    diary off
end
diaryname = ['Log_' strrep(strrep(...
datestr(now),' ','_'),':','-') '.txt'];
diary(diaryname)
t = timer('timerfcn',@updatediary,...
'period',10,...
'ExecutionMode','fixedRate');
start(t)
%put program to run here
propagation

diary off
```

propagation:

```
addpath(genpath('../matconvnet-1.0-beta20'));
addpath(genpath('../minFunc'));
startTime = tic;
params = initParams();
```

```
CNN_features = forwardCNN(params);

% load("RNN_weights");

RNN_output = forwardRNN(CNN_features, params, rnn);

% load('Wcat');

[maximum, y_predict] = softmaxTest(Wcat, RNN_output);
load("classes");
disp('prediction')
disp(classes(y_predict))
toc(startTime)
```

initParams:

```
function params = initParams()
%option to display incorrect classifications
params.recogDisp = 1; %value other than 0 will display all misclassified objects

% Set the data folder here
params.dataFolder = '../input/';

% set the data split to train and test on
params.split = 2;

% set the number of CNN filters
params.numFilters = 256; %number of filters in selected layer of CNN model

% set the number of RNN to use
params.numRNN = 8;

%load the pretrained deep net model
run('vl_setupnn.m')
params.cnnModel.net = load('imagenet-vgg-f.mat');

%choose the layer in deep net to extract feature
params.layer = 13;

params.RFS = [13 13];
```

forwardCNN:

```
function [CNN_features] = forwardCNN(params)
%% init the data containers

instanceData = dir([params.dataFolder '/*.png']);
disp(instanceData)
fileImgName = [params.dataFolder  instanceData(1).name];
img = imread(fileImgName);
img = imresize(img, [224, 224]);
img = single(img);

fim = extract_Alexnet(img, params.cnnModel.net, params.layer);
CNN_features = permute(fim, [3, 2, 1]); %from 13x13x256 to 256x13x13

return

function fileBool = isValid(name)
fileBool = (~strcmp(name,'.') && ~strcmp(name,'..') && ~strcmp(name,'.DS_Store'));
return;


function data = cutData(data)
data.data = data.data(1:data.count,:,:,:);
data.extra = data.extra(1:data.count,:);
return
```

forwardRNN:

```
function [RNN_output] = forwardRNN(CNN_features, params, rnn)

disp('Forward Prop RNN...');
RNN_output = forward(CNN_features, rnn, params);

RNN_output = RNN_output(:,:)';

function rnnData = forward(data, rnn, params)
data = permute(data,[1 2 3]);
[numMaps, rows, cols] = size(data);

numImgs=1;

RFS = params.RFS;
numRNN = params.numRNN;
assert(rows == cols);

% ensure a balanced tree is possible with these sizes
```

```
assert(mod(log(rows)/log(params.RFS(1)),1) < 1e-15);
assert(mod(log(cols)/log(params.RFS(2)),1) < 1e-15);

rnnData = zeros(numRNN, numMaps, numImgs);
for r = 1:numRNN
    if mod(r,8)==0
        disp(['RNN: ' num2str(r)]);
    end
    W = squeeze(rnn{1}.W(r,:,:));

    tree = data;

    tree = tanh(W * reshape(tree,[],1));

    rnnData(r,:) = squeeze(tree);
end

rnnData = permute(rnnData, [3 1 2]);
```

softmaxTest:

```
function [maximum, y_predict] = softmaxTest(theta,x)

    num = exp(theta*x);
    value = num/sum(num);
    [maximum, y_predict] = max(value);
end
```

Bibliography

[1] H. Heidenreich, "What are the types of machine learning? machine learning for the average person: An analysis of the types of machine learning, written for the average person.." https://towardsdatascience.com/what-are-the-types-of-machine-learning-e2b9e5d1756, 4 December 2018.

[2] "Teoria di darwin." https://www.chimica-online.it/biologia/teoria-di-darwin.htm.

[3] banana image, "Wallpapersdsc.net." https://www.wallpapersdsc.net/other/bananas-78081.html.

[4] lemon image, "Golden acre wines." https://www.goldenacrewines.co.uk/product/lemon-single/#.

[5] lemon slice image, "Legacy wine and spirits." https://www.legacylr.com/FRESH-PRODUCE-Fruits-Lemon-%28single%29-413609/.

[6] lemons image, "Brittanica." https://www.britannica.com/plant/lemon.

[7] K. Lai, L. Bo, X. Ren, and D. Fox, "Unsupervised feature learning for 3d scene labeling," in *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1817–1824, IEEE, May 2014.

[8] H. M. Bui, M. Lech, E. Cheng, K. Neville, and I. S. Burnett, "Object recognition using deep convolutional features transformed by a recursive network structure," *IEEE Access*, vol. 4, pp. 10059–10066, 2016.

[9] Wikipedia contributors, "Convolutional neural network — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Convolutional_neural_network&oldid=970121333, 2020. [Online; accessed 31-July-2020].

[10] Wikipedia contributors, "Recursive neural network — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Recursive_neural_network&oldid=950826199, 2020. [Online; accessed 31-July-2020].

[11] S. Marsland, *Machine learning: an algorithmic perspective*. CRC press, 2015.

[12] T. M. Mitchell *et al.*, "Machine learning. 1997," *Burr Ridge, IL: McGraw Hill*, vol. 45, no. 37, pp. 870–877, 1997.

[13] Wikipedia contributors, "Imagenet — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=ImageNet&oldid=967492644, 2020. [Online; accessed 2-August-2020].

[14] "Different types of deep learning models explained." https://roboticsbiz.com/different-types-of-deep-learning-models-explained/, 15 March 2020.

[15] A. Al-Masri, "What are overfitting and underfitting in machine learning?." https://towardsdatascience.com/what-are-overfitting-and-underfitting-in-machine-learning-a96b30864690, 22 June 2019.

[16] T. Shah, "About train, validation and test sets in machine learning." https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7, 06 December 2017.

[17] "Overfitting in machine learning: What it is and how to prevent it." https://elitedatascience.com/overfitting-in-machine-learning.

[18] J. Brownlee, "Difference between a batch and an epoch in a neural network." https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/, 20 July 2018.

[19] "Early stopping. terminate a training session given certain conditions." https://deeplearning4j.konduit.ai/tuning-and-training/early-stopping.

[20] T. U. of Queensland, "Action potentials and synapses." https://qbi.uq.edu.au/brain-basics/brain/brain-physiology/action-potentials-and-synapses#:~:text=Key.

[21] J. Calbet, "Hebb's rule with an analogy. psychology and neuroscience." https://neuroquotient.com/en/pshychology-and-neuroscience-hebb-principle-rule/, 14 March 2018.

[22] S. Sharma, "Activation functions in neural networks." https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6, 06 September 2017.

[23] D. Gupta, "Fundamentals of deep learning – activation functions and when to use them?." https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/#:~:text=ReLU, 30 January 2020.

[24] "A beginner's guide to multilayer perceptrons (mlp)." https://pathmind.com/wiki/multilayer-perceptron.

[25] G. Giacoppo, "Introduzione alle convolutional neural networks." https://medium.com/@giuseppegiacoppo/introduzione-alle-convolutional-neural-networks-602932527e56, 02 March 2019.

[26] W. Koehrsen, "Recurrent neural networks by example in python." https://towardsdatascience.com/recurrent-neural-networks-by-example-in-python-ffd204f99470, 05 November 2018.

[27] P. Eising, "What exactly is an api?." https://medium.com/@perrysetgo/what-exactly-is-an-api-69f36968a41f, 07 December 2017.

[28] "About keras." https://keras.io/about/.

[29] "Shear." https://mathworld.wolfram.com/Shear.html.

[30] C. Maklin, "Dropout neural network layer in keras explained." https://towardsdatascience.com/machine-learning-part-20-dropout-keras-layers-explained-8c9f6dc4c9ab#:~:text=Dropout., 03 June 2019.

[31] J. Brownlee, "A gentle introduction to pooling layers for convolutional neural networks." https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/#:~:text=Maximum., 05 July 2019.

[32] F. Chollet, "Building powerful image classification models using very little data." https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html, 05 June 2016.

[33] "What it is matlab?." https://www.mathworks.com/videos/matlab-overview-61923.html.

[34] P. Wozniak, H. Afrisal, R. G. Esparza, and B. Kwolek, "Scene recognition for indoor localization of mobile robots using deep cnn," in *International Conference on Computer Vision and Graphics*, pp. 137–147, Springer, 2018.

[35] A. (https://stats.stackexchange.com/users/64423/azrael), "Recurrent vs recursive neural networks: Which is better for nlp?." Cross Validated. URL:https://stats.stackexchange.com/q/158995 (version: 2015-06-27).

[36] R. Socher, C. C. Lin, C. Manning, and A. Y. Ng, "Parsing natural scenes and natural language with recursive neural networks," in *Proceedings of the 28th international conference on machine learning (ICML-11)*, pp. 129–136, 2011.

[37] R. Krzaczyński, "Limited-memory broyden-fletcher-goldfarb-shanno algorithm in ml.net." `https://towardsdatascience.com/limited-memory-broyden-fletcher-goldfarb-shanno-algorithm-in-ml-net-118dec066ba` 08 March 2020.