Table of Contents

Table of Figures

Chapter One: Introduction

1.1 Autonomous vehicles

1.1.1 Road Accidents

It is worth asking at this point whether distrust of autonomous technology has a semblance of the truth and is it backed by some authoritative research. According to the WHO, approximately 1 million people die due to road accidents every year. And most of the accidents are caused due to avoidable reasons. Considering the fact that more than 90% of fatal car accidents are caused due to human negligence, the assertion that autonomous cars would not lead to a significant decline in the number of car crashes is simply absurd and defies even the most elementary logic.

1.1.2 Solving Problem by Autonomous Cars

With autonomous cars, there is no possibility of drunken driving, dizziness or the driver getting drowsy, which causes a lot of accidents. Removing the element of human error margin would undoubtedly lead to a marked decline in the number of accidents and make the roads safer and save many lives.

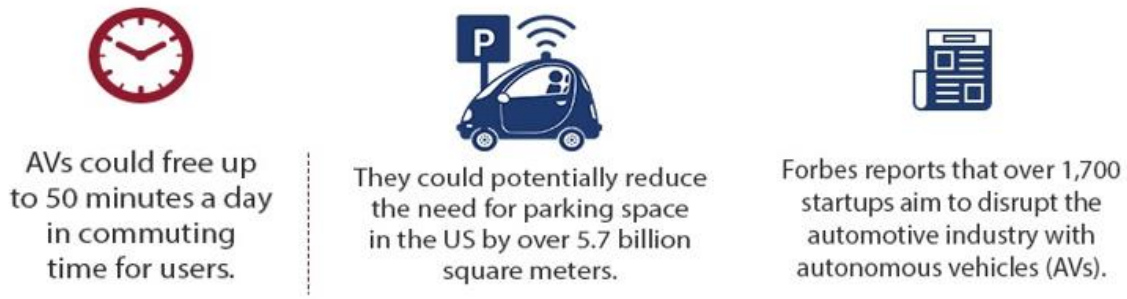


Figure 1

Also, in the case of autonomous car collisions, studies have found out that it not often the case of a technical glitch or system malfunction, but humans darting across the streets, changing lanes suddenly, texting while walking and not observing all the traffic rules, or simply being stupefied by the sight of an autonomous vehicle, in the future, as artificial intelligence evolves, autonomous cars would be able to predict the behavioral patterns of the pedestrians and prepare themselves accordingly.

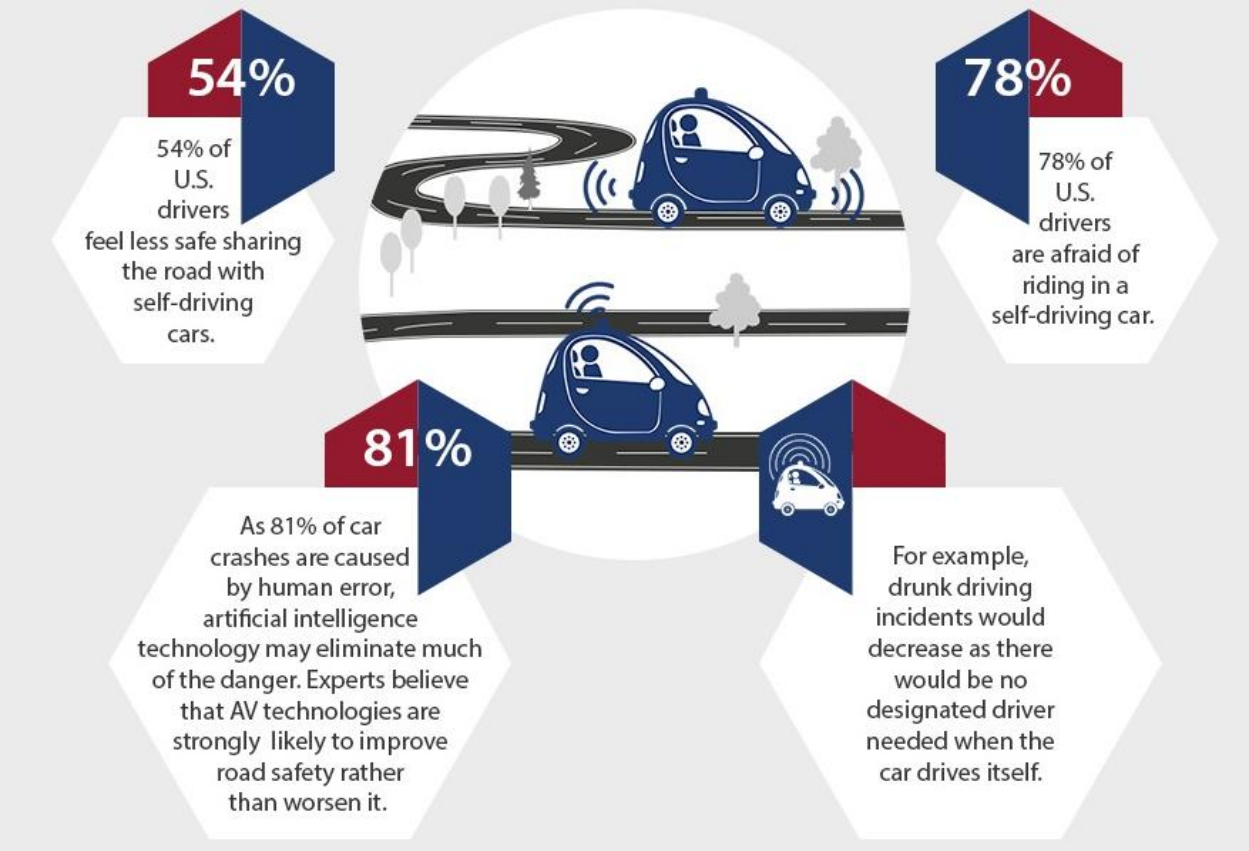


Figure 2

1.1.3 Autonomous Car

An autonomous car is a vehicle that can guide itself without human conduction. This kind of vehicle has become a concrete reality and may pave the way for future systems where computers take over the art of driving. An autonomous car is also known as a driverless car, robot car, self-driving car or autonomous vehicle.

1.1.3.1 Autonomous Car Components

The monitoring system of autonomous cars that consists of powerful sensors and LiDAR's view the objects ahead, with much more clarity, precision and a 360-degree view than the human eye and then chart the route

of the car accordingly. While autonomous cars are good at identifying hurdles ahead, it cannot distinguish between pedestrians and inanimate objects.

vehicles? Nevertheless, the metrics of safety considerations cannot overrule autonomous cars because, in any given time and situation, they are statistically safer than human-driven cars, and intelligent AI-powered systems are immune to the foibles and temperamental indiscretions of humans.

What is needed is setting public expectations and gradually erasing doubts that the citizens have. Along with companies like Waymo, governments should also step in to allay the fears and disseminate the advantages of autonomous cars — that range from reduction in fuel consumption, less traffic congestion to the decreased probability of a collision. A framework conducive to emerging enterprises and minimal regulations that promote and incentivize innovation is needed, in addition to a dedicated task force and participatory approach between governments and private industry. No system is engineered to be free from the probability of error margin. The error margin can only be made negligible and then the cause of the error can be identified and rectified with a permanent effect. In autonomous cars, because of extensive research and cutting-edge technical equipment used, the error margin is already very low, and it is possible that it may be obliterated altogether once the test phase is over and the technology incorporates new innovations. Considering the enormous potential of autonomous car technology and the benefits it has in store for the society, to write it off and spell doom is akin to throwing the baby along with the bathwater.

Self-driving cars might seem like a recent thing, but experiments of this nature have been taking place for nearly 100 years. In fact, centuries earlier, Leonardo Da Vinci designed a self-propelling cart hailed by some as the world's first robot. At the World's Fair in 1939, a theatrical and industrial designer named Norman Bel Geddes put forth a ride-on exhibit called Futurama, which depicted a city of the future featuring automated highways. However, the first self-driving cars didn't arrive until a series of projects in the 80s undertaken by Carnegie Mellon University, Bundeswehr University, and Mercedes-Benz. The Eureka Prometheus Project proposed an automated road system in which cars moved independently, with cities linked by vast expressways. At the time, it was the largest R&D project ever in the field of driverless cars.

Since 2013, US states Nevada, Florida, California, and Michigan have all passed laws permitting autonomous cars; more will surely follow. Autonomous vehicle components. Ouster's OS-1 3D LiDAR sensor captures point cloud data and camera-like images to help autonomous vehicles 'see' their environment How does an autonomous vehicle operate and make sense of what it sees? It comes down to a

powerful combination of technologies, which can be roughly divided into hardware and software. Hardware allows the car to see, move and communicate through a series of cameras, sensors and V2V/V2I technology, while software processes information and informs moment-by-moment decisions, like whether to slow down. If hardware is the human body, software is the brain.

At Level Five Supplies, our tech is broadly categorized as follows:

- Data storage
- Drive-by-wire
- Positioning
- Power
- Processing
- Sensors
    - Camera
    - LiDAR
    - Radar
    - Ultrasonic
- Software

Autonomous vehicles rely on sophisticated algorithms running on powerful processors. These processors make second-by-second decisions based on real-time data coming from an assortment of sensors. Millions of test miles have refined the technology and driven considerable progress – but there is still a way to go.Driverless car technology companies the race to be the first self-driving car on the road is heating up. Level 5 autonomy is still some time away, but there is plenty else happening in the autonomous space, with aspects of driverless tech already making an appearance in today's mass-produced cars. Early adopters will enjoy benefits such as automatic parking and driving in steady, single-lane traffic.

1.1.3.2 Manufacturing Companies

- Google – presently leading the charge via Waymo, its self-driving subsidiary
- Tesla – models are now being fitted with hardware designed to improve Tesla Autopilot
- Baidu – in the process of developing Level 4 automated vehicles
- General Motors – developed the first production-ready autonomous vehicle
- Toyota – working with ride-hailing service Uber to bring about autonomous ridesharing

- Nvidia – created the world's first commercially available Level 2+ system
- Ford – planning to have their fully autonomous vehicle in operation by 2021
- nuTonomy – first company to launch a fleet of self-driving taxis in Singapore
- BMW – has teamed up with Intel and Mobileye to release a fully driverless car by 2021
- Oxbotica – will begin trialing autonomous cars in London in December 2019

Challenges in autonomous vehicle testing and validation

As we can see, there are many different autonomous vehicle projects in various stages of development. But there's a big difference between creating a test vehicle that'll run in reasonably tame conditions and building a multi-million strong fleet of cars that can handle the volatility and randomness of the real world. One of the biggest challenges is putting the computer in charge of everything, including exception handling. By exceptions, we mean variable circumstances like bad weather, traffic violations and environmental hazards. Fog, snow, a deer leaping into the road – how does a fully autonomous vehicle interpret and react to these situations? When we take the driver out of the picture completely, automation complexity soars compared to lower-level systems. The software must handle everything. Rigorous testing is essential, but it won't be enough on its own. Alternative methods such as simulation, bootstrapping, field experience and human reviews will also be necessary to ensure the safety of the vehicle.

For the time being, it means implementing each new capability carefully and gradually: hence why it will be a good few years before we see Level 5 vehicles on the road. Global perceptions of autonomous car technology How does the general public feel about the onset of autonomous cars? Generally, perception is positive, but there's still a way to go. Not everyone is convinced. The main bone of contention is safety. Autonomous car manufacturers must prove beyond doubt the safety of their vehicles before they can hope for widespread adoption. One survey conducted across China, Germany and the US found that drivers want to decide for themselves when to let a car drive autonomously and when to take over. The survey also found that trust in autonomous cars is nearly twice as high in China.A second survey discovered that the higher the level of automation, the higher the doubt. There are still reservations about giving the vehicle total control, despite having a positive view on autonomous vehicles generally. Another factor playing on people's minds is cybercrime: fear of personal data falling into the wrong hands. People are generally happy to share data for safety reasons, but less so when it ends up being sold to related service providers.

1.1.4 Availability of Autonomous Cars

If we're talking about self-driving features, then the future has arrived. There are already cars on the road with Advanced Driver Assistance Systems in play. But a fully autonomous vehicle that can encounter and navigate any driving scenario without human intervention won't be mainstream for a little while yet, mostly due to cost and regulations. Level 5 systems will have the ability to drive anywhere a human driver could. What we'll likely come across first are Level 4 vehicles where autonomy is confined to mapped zones. They may also be limited by certain weather conditions. Business estimates there will be 10 million self-driving cars will be on the road by 2020. Chances are you'll be able to ride in one long before you can buy one, possibly even within the year. As for Level 5, estimations range from 2021 onwards. The past decade has been a pivotal one for automobile technology. In the face of changing political and legal circumstances, the future of autonomous vehicles is uncertain, and adoption will be gradual as society adjusts to the changes – not to mention upgrading transport infrastructures and systems. But of one thing we can be sure: the automotive industry will never be the same again

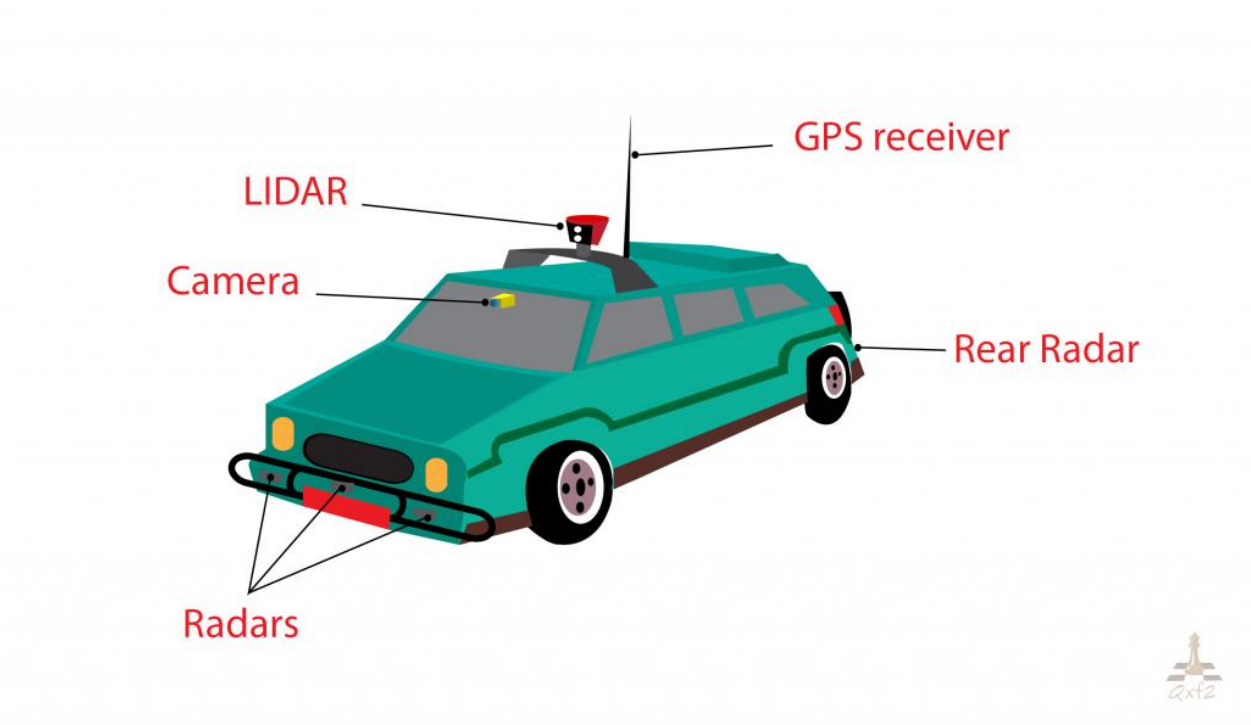


Figure 3

1.2. General Idea of The Project

In this chapter we will discuss a general idea about the project and what we want to achieve, and the steps done to achieve the project:

1.2.1 Machine Learning and Neural Networks

What is machine learning?

Machine learning is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. Machine learning focuses on the development of computer programs that can access data and use it learn for themselves.

What are neural networks?

A neural network is a network or circuit of neurons, or in a modern sense, an artificial neural network, composed of artificial neurons or nodes. Thus, a neural network is either a biological neural network, made up of real biological neurons, or an artificial neural network, for solving artificial intelligence (AI) problems. The connections of the biological neuron are modelled as weights. A positive weight reflects an excitatory connection, while negative values mean inhibitory connections. All inputs are modified by a weight and summed. This activity is referred as a linear combination. Finally, an activation function controls the amplitude of the output. For example, an acceptable range of output is usually between 0 and 1, or it could be −1 and 1.

These artificial networks may be used for predictive modelling, adaptive control and applications where they can be trained via a dataset. Self-learning resulting from experience can occur within networks, which can derive conclusions from a complex and seemingly unrelated set of information

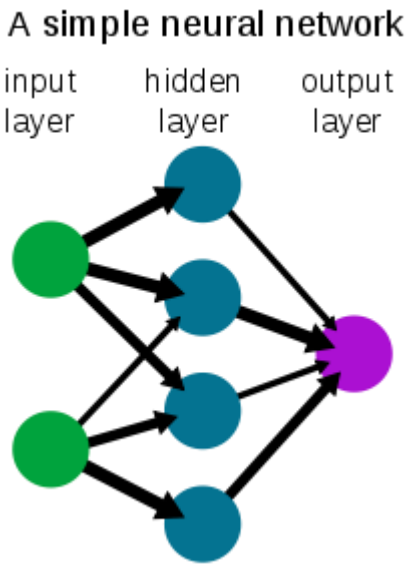


Figure 4

Why do we need neural networks in the project?

We will use a dataset of common objects we see normally in our daily life in streets to train our neural network to be able to recognise these objects in any future image that would be our input to the neural network.

Examples:

1. Cars
2. Human bodies
3. Cats
4. Dogs
5. Buses

6. Traffic posts

1.2.2 Phases of The Project

Our project consists mainly of 2 main phases:

1- 1st phase is to find a suitable architecture of a neural network that would be able to satisfy our constrains (delay, complexity, etc..) and implement this architecture using python machine learning modules like TensorFlow and PyTorch

2- 2nd phase after simulating our neural network we will get the weights resulted from the simulation (more details on that later) and use it to implement a hardware chip on a FPGA development kit

We decided to implement the project on a FPGA kit as hardware is a lot faster than software simulations and in object detection in autonomous cars we need very fast response as it is very critical as very small delays may lead to hazardous events also, we favoured it over ASIC due to reconfigurability

1.2.3 What Will be Discussed In the Following Chapters

We will use a top-down approach to discuss what we did through the project this semester and we will end it with a summary to all what we did through the semester and what we will do in the next semester and what tools we will need to learn(Future work)

Topics that will be discussed:

1- What are the types of neural networks and which one we will use

2- What are the mathematics and concepts behind the chosen neural network type

3-What are the problems faced with this type of network and how to solve them

4-What tools and technologies we used and we will need to do the neural network

5-A comparison between the different architectures of the type chosen

6-choosing the best architecture according to our constrains

7-Future work

Chapter two: Neural Networks

2.1 Neural Networks and Its types

Neural networks are a specific set of algorithms that have revolutionized the field of machine learning. They are inspired by biological neural networks and the current so-called deep neural networks have proven to work quite very well. Neural Networks are themselves general function approximations, that is why they can be applied to literally almost any machine learning problem where the problem is about learning a complex mapping from the input to the output space. Generally, these architectures can be put into 3 specific categories Feed-Forward Neural Networks: These are the commonest type of neural network in practical applications. The first layer is the input and the last layer is the output. If there is more than one hidden layer, we call them “deep” neural networks. They compute a series of transformations that change the similarities between cases. The activities of the neurons in each layer are a non-linear function of the activities in the layer below.

2.1.1 Recurrent Networks

A These have directed cycles in their connection graph. That means you can sometimes get back to where you started by following the arrows. They can have complicated dynamics, and this can make them very difficult to train. They are more biologically realistic. There is a lot of interest at present in finding efficient ways of training recurrent nets.

Recurrent neural networks are a very natural way to model sequential data. They are equivalent to very deep nets with one hidden layer per time slice; except that they use the same weights at every time slice, and they get input at every time slice. They can remember information in their hidden state for a long time but is very hard to train them to use this potential.

2.1.2 Symmetrically Connected Networks

These are like recurrent networks, but the connections between units are symmetrical (they have the same weight in both directions). Symmetric networks are much easier to analyze than recurrent networks. They are also more restricted in what they can do because they obey an energy function. Symmetrically connected nets without hidden units are called “Hopfield Nets.” Symmetrically connected networks with hidden units are called “Boltzmann machines." to understand RNNs, we need to have a brief overview of sequence modeling. When applying machine learning to sequences, we often want to turn an input sequence into an output sequence that lives in a different domain; for example, turn a sequence of sound pressures into a sequence of word identities. When there is no separate target sequence, we can get a teaching signal by trying to predict the next term in the input sequence. The target output sequence is the input sequence with an advance of 1 step. This seems much more natural than trying to predict one pixel in an image from the other pixels, or one patch of an image from the rest of the image. Predicting the next term in a sequence blurs the distinction between supervised and unsupervised learning. It uses methods designed for supervised learning, but it doesn’t require a separate teaching signal.

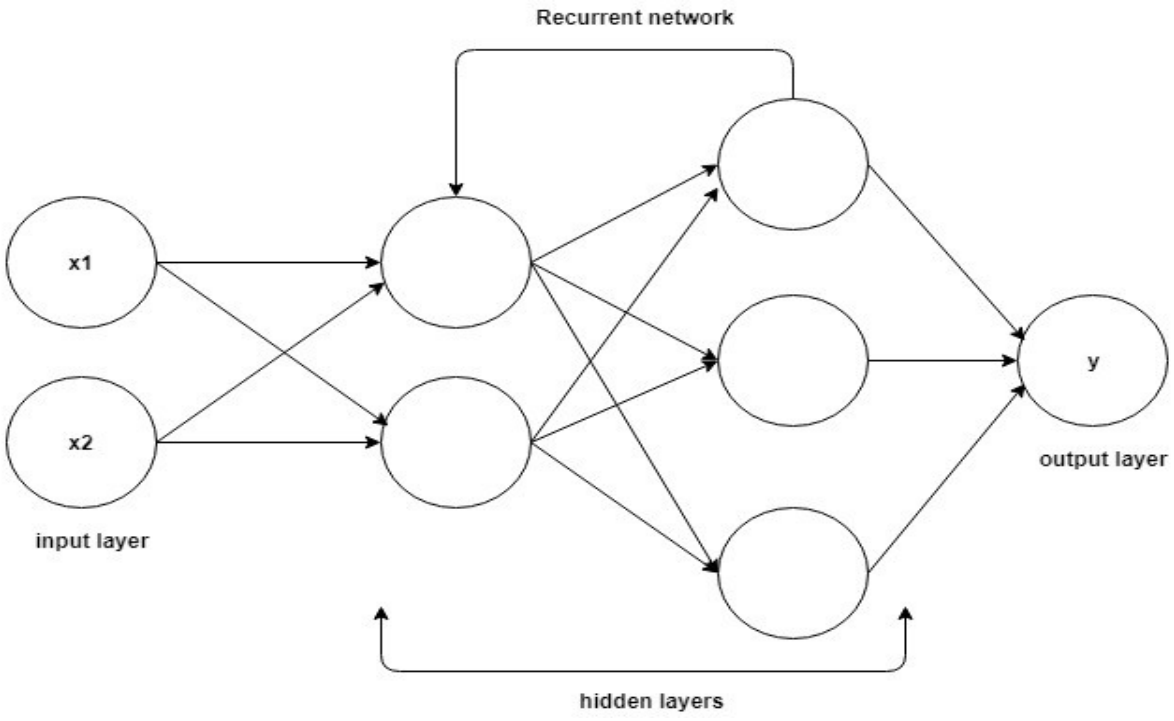


Figure 5

2.1.3 Hopfield Network

These are like recurrent networks, but the connections between units are symmetrical (they have the same weight in both directions). Symmetric networks are much easier to analyze than recurrent networks. They are also more restricted in what they can do because they obey an energy function. Symmetrically connected Hopfield net is composed of binary threshold units with recurrent connections between them. In 1982, John Hopfield realized that if the connections are symmetric, there is a global energy function. Each binary "configuration" of the whole network has an energy; while the binary threshold decision rule causes the network to settle to a minimum of this energy function. A neat way to make use of this type of computation is to use memories as energy minima for the neural net. Using energy minima to represent memories gives a content-addressable memory. An item can be accessed by just knowing part of its content. It is robust against hardware damage.

Each time we memorize a configuration, we hope to create a new energy minimum. But what if two nearby minima at an intermediate location? This limits the capacity of a Hopfield net. So how do we increase the capacity of a Hopfield net? Physicists love the idea that the math they already know might explain how the brain works. Many papers were published in physics journals about Hopfield nets and their storage capacity. Eventually, Elizabeth Gardner figured out that there was a much better storage rule that uses the full

capacity of the weights. Instead of trying to store vectors in one shot, she cycled through the training set many times and used the perceptron convergence procedure to train each unit to have the correct state given the states of all the other units in that vector. Statisticians call this technique “pseudo-likelihood.”

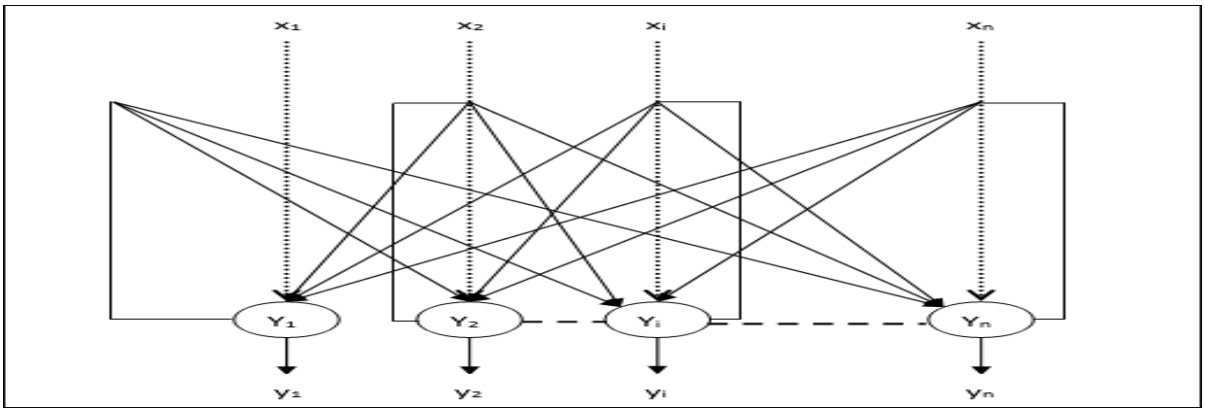


Figure 6

2.1.4 Boltzmann Machine Network

A Boltzmann machine is a type of stochastic recurrent neural network. It is the stochastic, generative counterpart of Hopfield nets. It was one of the first neural networks capable of learning internal representations and can represent and solve difficult combinatoric problems. The goal of learning for Boltzmann machine learning algorithm is to maximize the product of the probabilities that the Boltzmann machine assigns to the binary vectors in the training set. This is equivalent to maximizing the sum of the log probabilities that the Boltzmann machine assigns to the training vectors.

It is also equivalent to maximizing the probability that we would obtain exactly the N training cases if we did the following:

1) Let the network settle to its stationary distribution N different time with no external input; and

2) Sample the visible vector once each time.

Unfortunately, there is a serious practical problem with the Boltzmann machine, namely that it seems to stop learning correctly when the machine is scaled up to anything larger than a trivial machine.[citation needed] This is due to a number of effects, the most important of which are:

the time the machine must be run in order to collect equilibrium statistics grows exponentially with the machine's size, and with the magnitude of the connection strengths

connection strengths are more plastic when the units being connected have activation probabilities intermediate between zero and one, leading to a so-called variance trap. The net effect is that noise causes the connection strengths to follow a random walk until the activities saturate.

2.1.5 Deep Belief Network

Back-propagation is considered the standard method in artificial neural networks to calculate the error contribution of each neuron after a batch of data is processed. However, there are some major problems using backpropagation. Firstly, it requires labelled training data, while almost all data is unlabelled. Secondly, the learning time does not scale well, which means it is very slow in networks with multiple hidden layers. Thirdly, it can get stuck in poor local optima, so for deep nets they are far from optimal.

To overcome the limitations of backpropagation, researchers have considered using unsupervised learning approaches. This helps keep the efficiency and simplicity of using a gradient method for adjusting the weights, but also use it for modelling the structure of the sensory input. They adjust the weights to maximize the probability that a generative model would have generated the sensory input.

Convolutional Deep Belief Network (CDBN) is a type of deep artificial neural network that is composed of multiple layers of convolutional restricted Boltzmann machines stacked together. Alternatively, it is a hierarchical generative model for deep learning, which is highly effective in the tasks of image processing and object recognition, though it has been used in other domains too. The salient features of the model include the fact that it scales well to high-dimensional images and is translation-invariant.

CDBNs use the technique of probabilistic max pooling to reduce the dimensions in higher layers in the network. Training of the network involves a pre-training stage accomplished in a greedy layer-wise manner, like other deep belief networks. Depending on whether the network is to be used for discrimination or generative tasks, it is then "fine-tuned" or trained with either back-propagation or the up-down algorithm (contrastive-divergence), respectively.

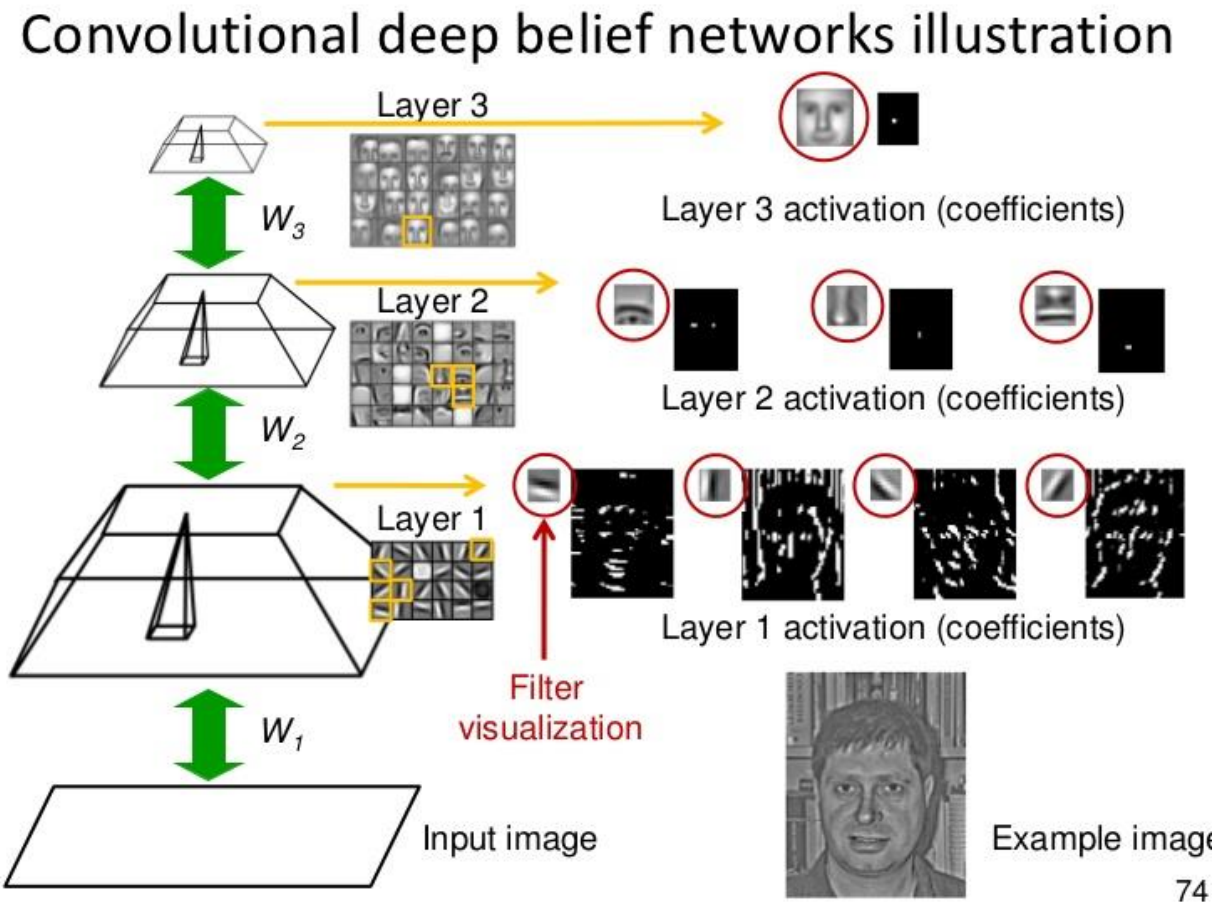


Figure 7

2.1.6 Convolutional neural network

2.1.6.1 Design

A convolutional neural network consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of a series of convolutional layers that convolve with a multiplication or other dot product. The activation function is commonly a RELU layer and is subsequently followed by additional convolutions such as pooling layers, fully connected layers and normalization layers, referred to as hidden layers because their inputs and outputs are masked by the activation function and final convolution. The final convolution, in turn, often involves backpropagation in order to more accurately weight the product. Though the layers are colloquially referred to as convolutions, this is only by convention. Mathematically, it is technically a sliding dot product or cross-correlation. This has significance for the indices in the matrix, in that it affects how weight is determined at a specific index point.

2.1.6.2 Convolutional

When programming a CNN, the input is a tensor with shape (number of images) x (image width) x (image height) x (image depth). Then after passing through a convolutional layer, the image becomes abstracted to a feature map, with shape (number of images) x (feature map width) x (feature map height) x (feature map channels). A convolutional layer within a neural network should have the following attributes: Convolutional kernels defined by a width and height (hyper-parameters).The number of input channels and output channels (hyper-parameter).The depth of the Convolution filter (the input channels) must be equal to the number channels (depth) of the input feature map. Convolutional layers convolve the input and pass its result to the next layer. This is like the response of a neuron in the visual cortex to a specific stimulus. Each convolutional neuron processes data only for its receptive field. Although fully connected feedforward neural networks can be used to learn features as well as classify data, it is not practical to apply this architecture to images. A very high number of neurons would be necessary, even in a shallow (opposite of deep) architecture, due to the very large input sizes associated with images, where each pixel is a relevant variable. For instance, a fully connected layer for a (small) image of size 100 x 100 has 10,000 weights for each neuron in the second layer. The convolution operation brings a solution to this problem as it reduces the number of free parameters, allowing the network to be deeper with fewer parameters. For instance, regardless of image size, tiling regions of size 5 x 5, each with the same shared weights, requires only 25 learnable parameters. In this way, it resolves the vanishing or exploding gradients problem in training traditional multi-layer neural networks with many layers by using backpropagation.

2.1.6.3 Pooling

Convolutional networks may include local or global pooling layers to streamline the underlying computation. Pooling layers reduce the dimensions of the data by combining the outputs of neuron clusters at one layer into a single neuron in the next layer. Local pooling combines small clusters, typically 2 x 2. Global pooling acts on all the neurons of the convolutional layer. In addition, pooling may compute a max or an average. Max pooling uses the maximum value from each of a cluster of neurons at the prior layer. Average pooling uses the average value from each of a cluster of neurons at the prior layer.

2.1.6.4 Fully connected

Fully connected layers connect every neuron in one layer to every neuron in another layer. It is in principle the same as the traditional multi-layer perceptron neural network (MLP). The flattened matrix goes through a fully connected layer to classify the images. In neural networks, each neuron receives input from some number of locations in the previous layer. In a fully connected layer, each neuron receives input from every

element of the previous layer. In a convolutional layer, neurons receive input from only a restricted subarea of the previous layer. Typically, the subarea is of a square shape (e.g., size 5 by 5). The input area of a neuron is called its receptive field. So, in a fully connected layer, the receptive field is the entire previous layer. In a convolutional layer, the receptive area is smaller than the entire previous layer.

2.1.6.5 Weights

Each neuron in a neural network computes an output value by applying a specific function to the input values coming from the receptive field in the previous layer. The function that is applied to the input values is determined by a vector of weights and a bias (typically real numbers). Learning, in a neural network, progresses by making iterative adjustments to these biases and weights. The vector of weights and the bias are called filters and represent features of the input (e.g., a shape). A distinguishing feature of CNNs is that many neurons can share the same filter. This reduces memory footprint because a single bias and a single vector of weights are used across all receptive fields sharing that filter, as opposed to each receptive field having its own bias and vector weighting.

2.2 Neural Networks Training

Once a network has been structured for an application, that network is ready to be trained. To start this process the initial weights are chosen randomly. Then, the training, or learning, begins.

There are two approaches to training - supervised and unsupervised. Supervised training involves a mechanism of providing the network with the desired output either by manually "grading" the network's performance or by providing the desired outputs with the inputs. Unsupervised training is where the network must make sense of the inputs without outside help.

The vast bulk of networks utilize supervised training. Unsupervised training is used to perform some initial characterization on inputs. However, in the full-blown sense of being truly self-learning, it is still just a shining promise that is not fully understood, does not completely work, and thus is relegated to the lab.

2.2.1 Supervised Training

In supervised training, both the inputs and the outputs are provided. The network then processes the inputs and compares its resulting outputs against the desired outputs. Errors are then propagated back through the system, causing the system to adjust the weights which control the network. This process occurs over and over as the weights are continually tweaked. The set of data which enables the training is called the "training set." During the training of a network the same set of data is processed many times as the connection weights are ever refined.

The current commercial network development packages provide tools to monitor how well an artificial neural network is converging on the ability to predict the right answer. These tools allow the training process to go on for days, stopping only when the system reaches some statistically desired point, or accuracy. However, some networks never learn. This could be because the input data does not contain the specific information from which the desired output is derived. Networks also don't converge if there is not enough data to enable complete learning. Ideally, there should be enough data so that part of the data can be held back as a test. Many layered networks with multiple nodes are capable of memorizing data. To monitor the network to determine if the system is simply memorizing its data in some nonsignificant way, supervised training needs to hold back a set of data to be used to test the system after it has undergone its training. (Note: memorization is avoided by not having too many processing elements.)

If a network simply can't solve the problem, the designer then has to review the input and outputs, the number of layers, the number of elements per layer, the connections between the layers, the summation, transfer, and training functions, and even the initial weights themselves. Those changes required to create a successful network constitute a process wherein the "art" of neural networking occurs.

Another part of the designer's creativity governs the rules of training. There are many laws (algorithms) used to implement the adaptive feedback required to adjust the weights during training. The most common technique is backward-error propagation, more commonly known as backpropagation. These various learning techniques are explored in greater depth later in this report.

Yet, training is not just a technique. It involves a "feel," and conscious analysis, to ensure that the network is not over trained. Initially, an artificial neural network configures itself with the general statistical trends of the data. Later, it continues to "learn" about other aspects of the data which may be spurious from a general viewpoint.

When finally, the system has been correctly trained, and no further learning is needed, the weights can, if desired, be "frozen." In some systems this finalized network is then turned into hardware so that it can be fast. Other systems don't lock themselves in but continue to learn while in production use.

2.2.2 Unsupervised Training.

The other type of training is called unsupervised training. In unsupervised training, the network is provided with inputs but not with desired outputs. The system itself must then decide what features it will use to group the input data. This is often referred to as self-organization or adaption.

At the present time, unsupervised learning is not well understood. This adaption to the environment is the promise which would enable science fiction types of robots to continually learn on their own as they encounter new situations and new environments. Life is filled with situations where exact training sets do not exist. Some of these situations involve military action where new combat techniques and new weapons might be encountered. Because of this unexpected aspect to life and the human desire to be prepared, there continues to be research into, and hope for, this field. Yet, at the present time, the vast bulk of neural network work is in systems with supervised learning. Supervised learning is achieving results.

One of the leading researchers into unsupervised learning is Tuevo Kohonen, an electrical engineer at the Helsinki University of Technology. He has developed a self-organizing network, sometimes called an auto-associator, that learns without the benefit of knowing the right answer. It is an unusual looking network in that it contains one single layer with many connections. The weights for those connections must be initialized and the inputs must be normalized. The neurons are set up to compete in a winner-take-all fashion.

Kohonen continues his research into networks that are structured differently than standard, feedforward, back-propagation approaches. Kohonen's work deals with the grouping of neurons into fields. Neurons within a field are "topologically ordered." Topology is a branch of mathematics that studies how to map from one space to another without changing the geometric configuration. The three-dimensional groupings often found in mammalian brains are an example of topological ordering.

Kohonen has pointed out that the lack of topology in neural network models make today's neural networks just simple abstractions of the real neural networks within the brain. As this research continues, more powerful self-learning networks may become possible. But currently, this field remains one that is still in the laboratory.

2.3 Mathematics of Neural Networks.

2.3.1 How Mathematically Do Neural Networks Work?

- A neuron is just a mathematical function that takes inputs (the outputs of the neurons pointing to it) and returns outputs.

- These outputs serve as inputs for the next layer, and so on until we get to the final, output layer, which is the actual value we return.

- There is an input layer, where each neuron will simply return the corresponding value in the inputs vector.
- For each set of inputs, the Neural Network's goal is to make each of its outputs **as close as possible to the actual expected values.**
- If we take 100x100px pictures of animals as inputs, then our input layer will have 30000 neurons. That's 10000 for all the pixels, times three since a pixel is already a triple vector (RGB values).
- We will then run the inputs through each layer. We get a **new vector as each layer's output, feed it to the next layer as inputs**, and so on.
- Each neuron in a layer will return a single value, so a layer's output vector will have as many dimensions as the layer has neurons.

- The following figure illustrates the main idea.

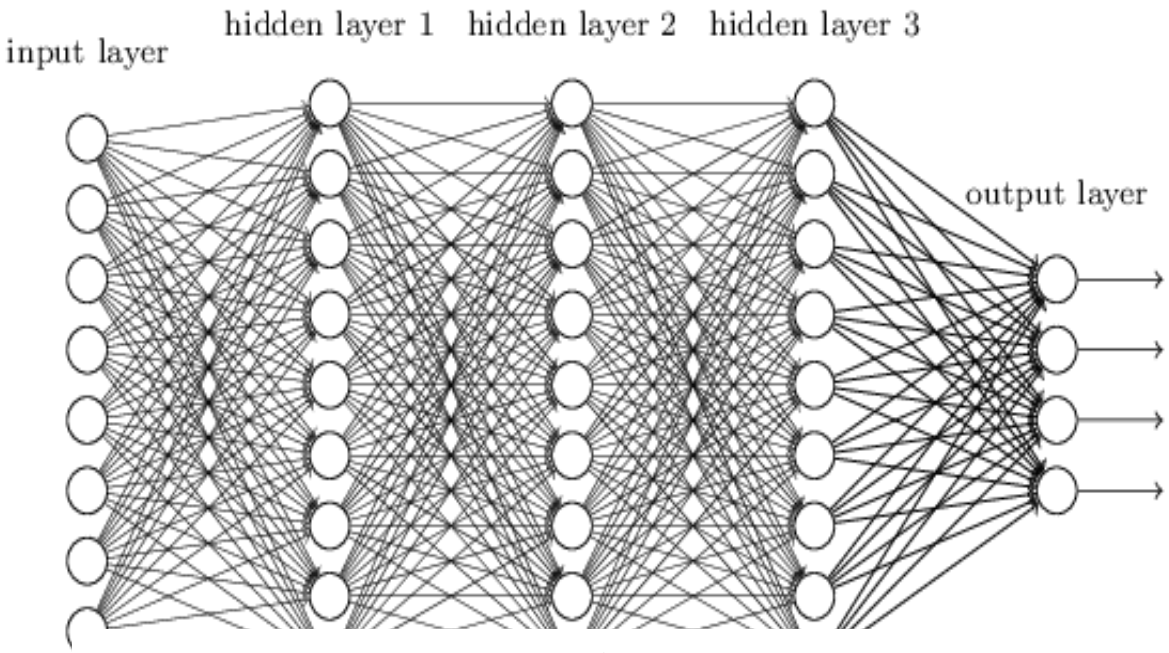


Figure 8

2.3.2 What does a Neuron do?

A neuron will take an input vector, and do three things to it:

1) Multiply it by a weights vector.

2) Add a bias value to that product.

3) Apply an activation function to that value.

- We'll typically use non-linear functions as activation functions**.** This is because the linear part is already handled by the previously applied product and addition.

2.3.2.1 Weight

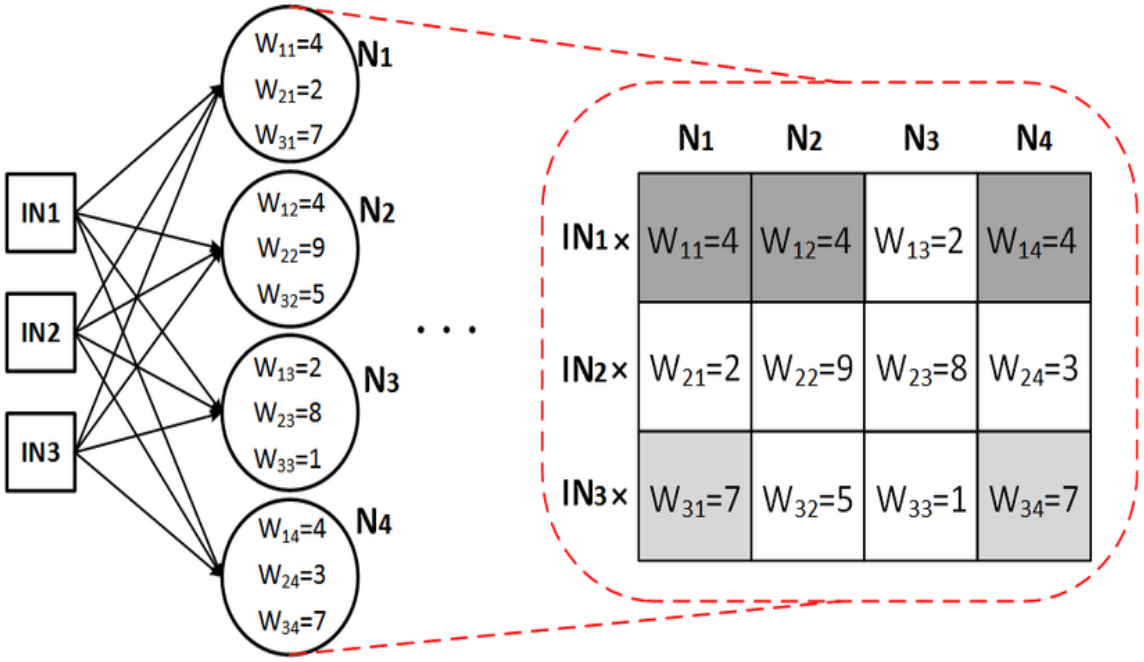


Figure 9

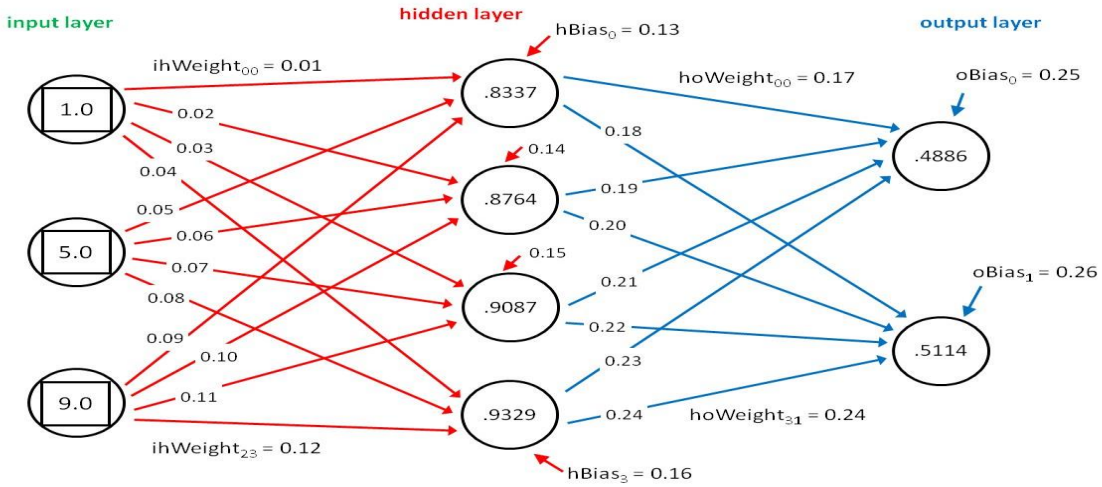
As you can see in the image, the input layer has 3 neurons and the very next layer (a hidden layer) has 4. We can create a matrix of 3 rows and 4 columns and insert the values of each weight in the matrix as done above. This matrix would be called **W1**. In the case where we have more layers, we would have more weight matrices, W2, W3, etc.
In general, if a layer L has N neurons and the next layer L+1 has M neurons, the **weight matrix** is an N-by-M matrix (N rows and M columns).

Again, look closely at the image, you'd discover that the largest number in the matrix is ***W22*** which carries a value of ***9***. Our ***W22*** connects ***IN2*** at the input layer to ***N2*** at the hidden layer. This means that ***"at this state"*** or ***currently***, our ***N2*** thinks that the input ***IN2*** is the most important of all 3 inputs it has received in making its own tiny decision.

2.3.2.2 Bias.

The bias is also a weight. Imagine you're thinking about a situation (trying to decide). You must think about all possible (or observable) factors. But what about parameters you haven't come across? What about factors you haven't considered? In a Neural Net, we try to cater for these unforeseen or non-observable factors. This is the bias. Every neuron that is not on the input layer has a bias attached to it, and the bias, just like the weight, carries a value. The image below is a good illustration.

Looking carefully at the layer in the hidden and output layers (with 4 and 2 neurons respectively), you'll find that each neuron has a tiny red/blue arrow pointing at it. You'll also discover that these tiny arrows have no source neuron.

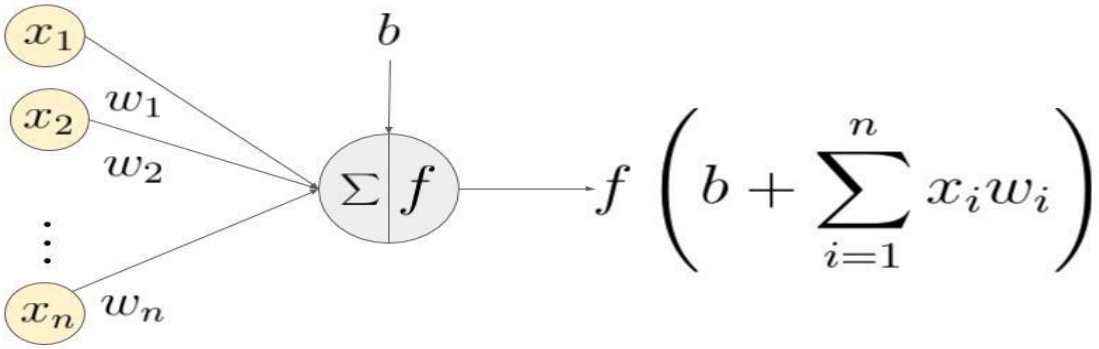


Just like weights can be viewed as a matrix, biases can also be matrices with 1 column (a **vector** if you please). As an example, the bias for the hidden layer above would be expressed as [[0.13], [0.14], [0.15], [0.16]].

Figure 10

2.3.2.3 Activation function.

For this section, let's focus on a single neuron. After aggregating all the input into it, a neuron is supposed to make a **tiny** decision on that output and return another output. This process (or function) is called an activation function.



An example of a neuron showing the input (x_1 - x_n), their corresponding weights (w_1 - w_n), a bias (b) and the activation function f applied to the weighted sum of the inputs.

Figure 11

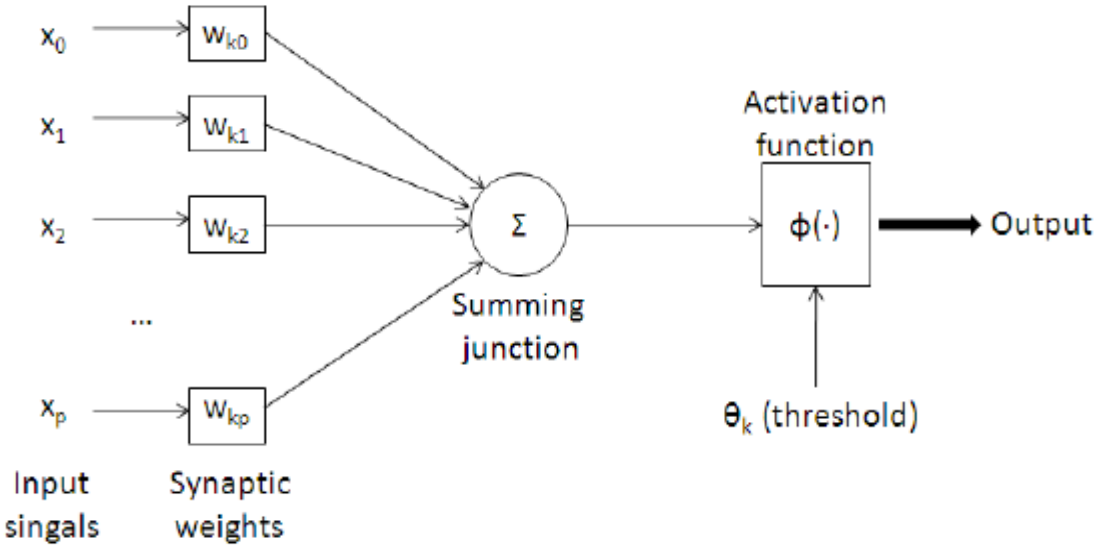


Figure 12

2.3.3 Loss Function Brief Intro

Loss function helps in optimizing the parameters of the neural networks. Our objective is to minimize the loss for a neural network by optimizing its parameters (weights). The loss is calculated using loss function by matching the target (actual) value and predicted value by a neural network. Then we use the gradient descent method to update the weights of the neural network such that the loss is minimized. This is how we train a neural network.

2.3.3.1 Mean Squared Error

When we have a regression task, one of the loss functions you can go ahead is this one. As the name suggests, this loss is calculated by taking the mean of squared differences between actual (target) and predicted values.

For Example, we have a neural network which takes house data and predicts house price. In this case, you can use the MSE loss. Basically, in the case where the output is a real number, you should use this loss function.

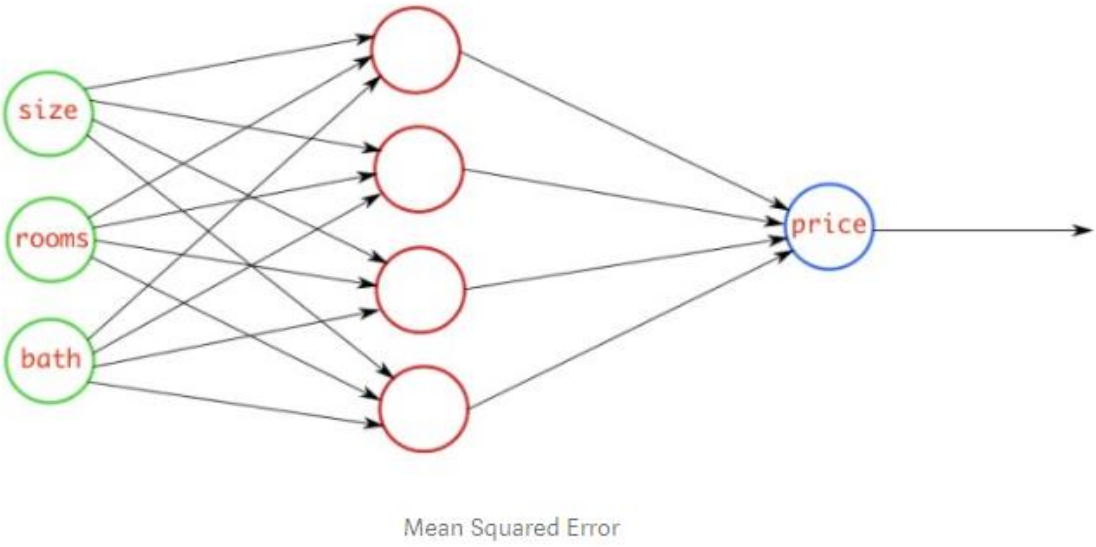


Figure 13

2.3.3.2 Binary Cross entropy

When we have a binary classification task, one of the loss function you can go ahead is this one. If you are using BCE loss function, you just need one output node to classify the data into two classes. The output value should be passed through a *sigmoid* activation function and the range of output is (0 – 1).

For example, we have a neural network which takes atmosphere data and predicts whether it will rain or not. If the output is greater than 0.5, the network classifies it as rain and if the output is less than 0.5, the network classifies it as not rain. (It could be opposite depending upon how you train the network). The More the probability score value, more the chance of raining.

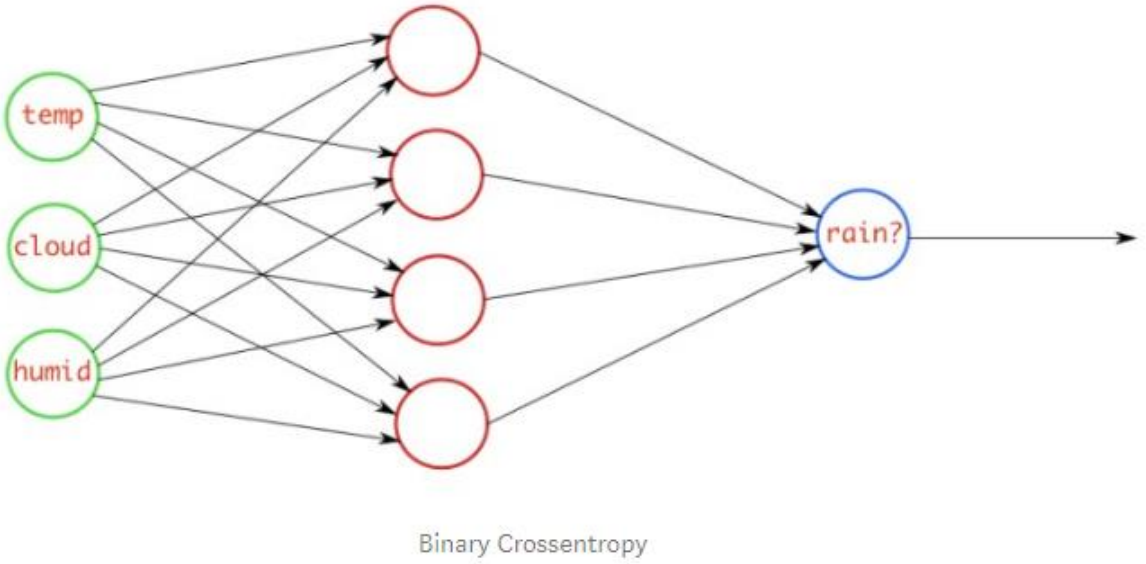


Figure 14

2.3.3.3 Categorical Cross entropy

When we have a multi-class classification task, one of the loss function you can go ahead is this one. If you are using `CCE` loss function, there must be the same number of output nodes as the classes. And the final layer output should be passed through a *SoftMax* activation so that each node output a probability value between (0–1).

For example, we have a neural network which takes an image and classifies it into a cat or dog. If cat node has high probability score, then the image is classified into cat otherwise dog. Basically, whichever class node has the highest probability score, the image is classified into that class.

2.3.4 Back

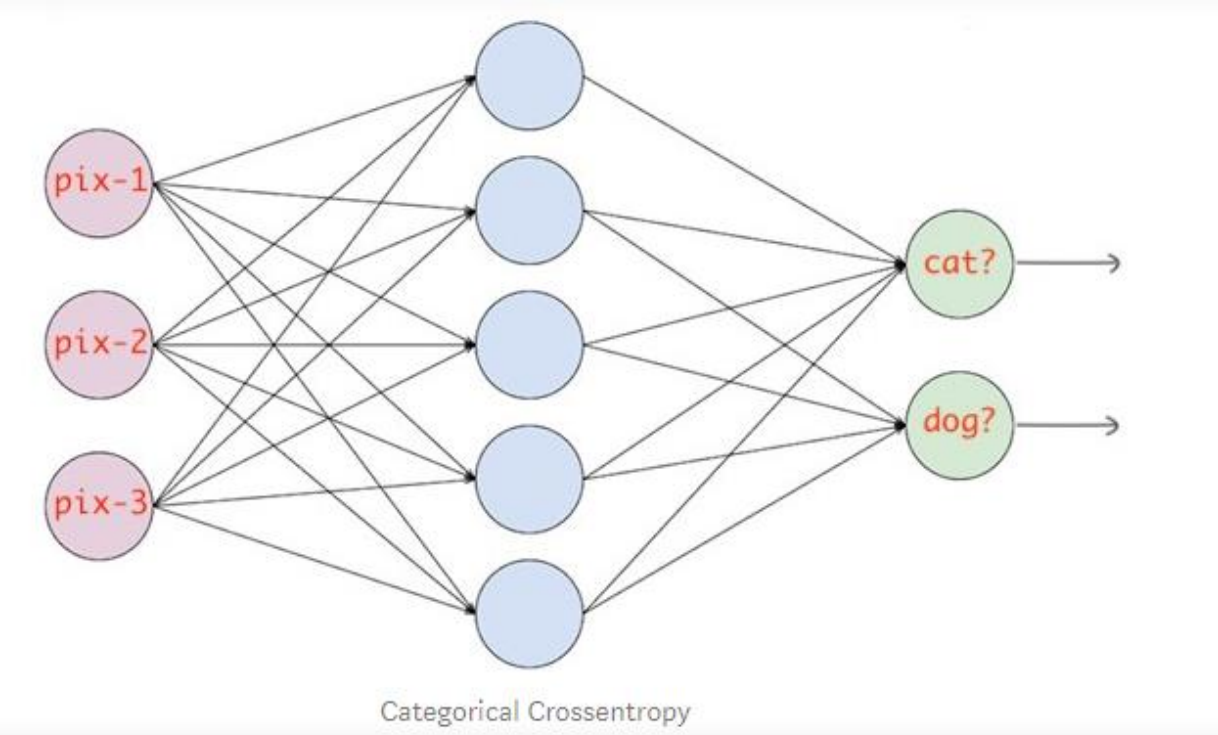


Figure 15

propagation algorithm Proof

Symbol	Meaning
x	Input
y	Output of the corresponding neuron
w	Weight
i	Index
E	Mean Square Error
t	Target value

Transfer potential (p) = ∑ xi × wi (which is simply the input scaled by a certain weight)

$\therefore \frac{dp}{d(wi)}$ = xi

E = 1/2 $(ti - yi)^2$ (if *yi* = *ti*, then E = 0 and we have the best case scenario)

$\frac{dE}{d(yi)}$ = - (ti-yi)

Assume we use sigmoid as an activation function

∴ yi = $\frac{1}{1+e^{-p}}$ (in which the input to the activation function (sigmoid) is the transfer potential and the output is yi)

$\frac{d(\text{yi})}{dp} = \frac{e^{-p}}{(1+e^{-p})^2} = \frac{e^{-p}}{1+e^{-p}} * \frac{1}{1+e^{-p}}$ = yi * $\frac{e^{-p}}{1+e^{-p}}$ = yi (1-yi)

By using the chain rule to find ($\frac{dE}{d\text{p}}$) and ($\frac{dE}{d(\text{wi})}$)

∴ $\frac{dE}{d\text{p}} = \frac{d(\text{yi})}{dp} * \frac{dE}{d(yi)}$ = - (ti-yi) (yi) (1-yi)

∴ $\frac{dE}{d(\text{wi})} = \frac{dE}{d\text{p}} * \frac{dp}{d(wi)}$ = - (xi) (yi) (1-yi) (ti-yi) = Δw

∴ wi = wi +Δw

So, in every iteration the value of each corresponding weight updates by a certain amount (Δw) in order to reduce the error, where Δw depends on the input, output and the target value.

- But what about the beginning? How should we initialize the weights at the first time? , if we initialize the weights randomly the neural network output may take a very large amount of time to converge to the right value and may even fail to converge so there must be a way of using the weights at the first time to minimize as possible the time of convergence and increase the probability of approaching the right choice This is usually done by a method known as Xavier initialization, the main goal of this method is to make the variance of the input and the output of the neurons almost the same.
- So, we pick the weights from a Gaussian distribution with zero mean and a variance of $\frac{1}{N}$, where N specifies the number of input neurons. so var (wi) = $\frac{1}{N}$

2.3.5 Types of Activation functions.

2.3.5.1 Step Function.

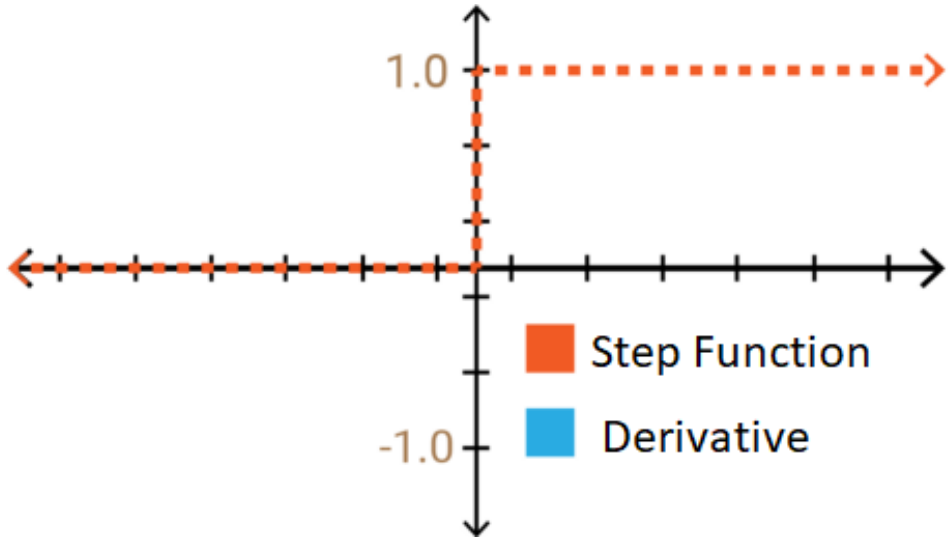


Figure 16

- It is a function that takes a binary value and is used as a binary classifier. Therefore, it is generally preferred in the output layers. It is not recommended to use it in hidden layers because it does not represent derivative learning value and it will not appear in the future.

2.3.5.2 Sigmoid Function

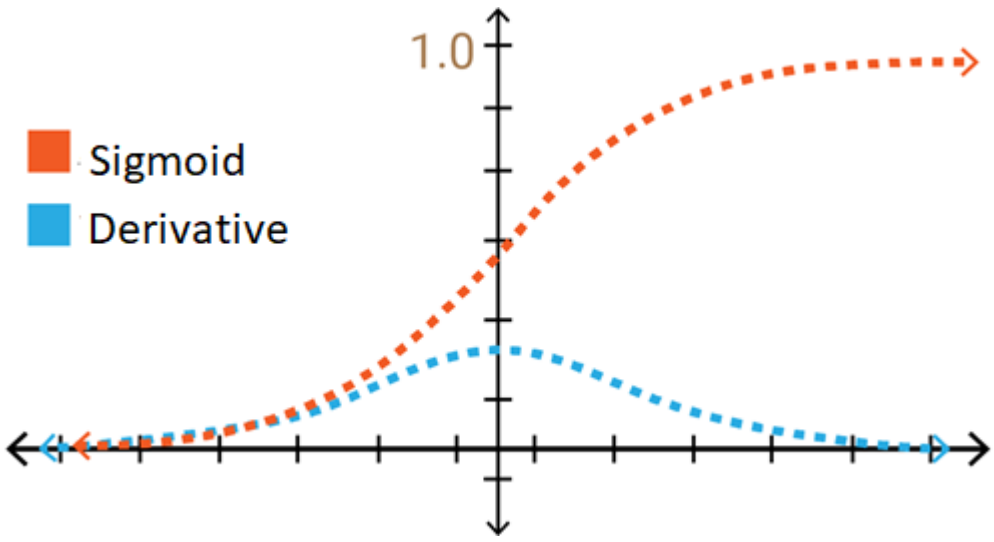


Figure 17

- So, let's think of non-binary functions. It is also derivated because it is different from the step function. This means that learning can happen. If we examine the graph x is between -2 and +2, y values change quickly. Small changes in x will be large in y. This means that it can be used as a good classifier. Another advantage of this function is that it produces a value in the range of (0, 1) when encountered with (- infinite, + infinite) as in the linear function. So the activation value does not vanish**.**

- So, what's the problem with sigmoid function? If we look carefully at the graph towards the ends of the function, y values react very little to the changes in x. Let's think about what kind of problem it is the derivative values in these regions are very small and converge to 0. This is called the vanishing gradient and the learning is minimal. if 0, not any learning! When slow learning occurs, the optimization algorithm that minimizes error can be attached to local minimum values and cannot get maximum performance from the artificial neural network model.

2.3.5.3 Hyperbolic Tangent Function.

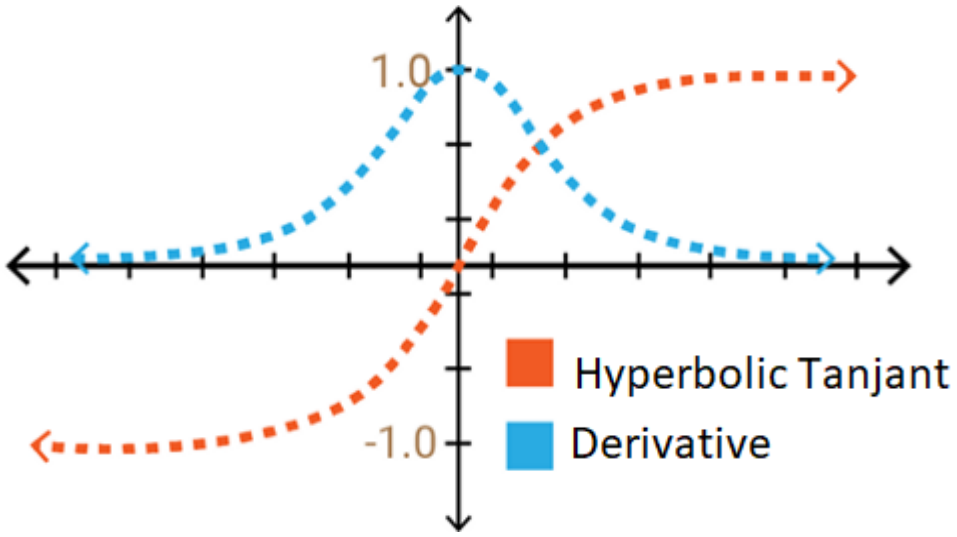


Figure 18

- It has a structure very similar to sigmoid function. However, this time the function is defined as (-1, + 1). The advantage over the sigmoid function is that its derivative is steeper, which means it can get more value. This means that it will be more efficient because it has a wider range for faster learning and grading. But again, the problem of gradients at the ends of the function continues.

2.3.5.4 Relu (Rectified Linear Unit)

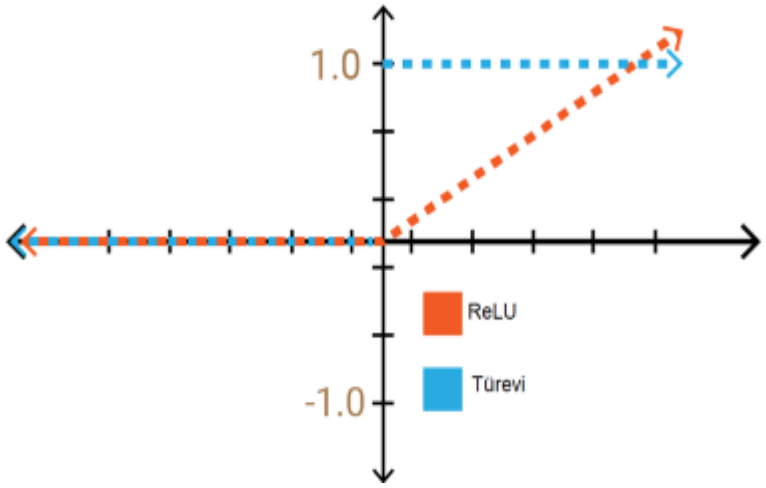


Figure 19

- At first glance, it will appear to have the same characteristics as the linear function on the positive axis. But above all, ReLU is not linear in nature
- Let's imagine a large neural network with too many neurons. Sigmoid and hyperbolic tangent caused almost all neurons to be activated in the same way. This means that the activation is very intensive. Some of the neurons in the network are active, and activation is infrequent, so we want an efficient computational load. We get it with ReLU. **T**he fact that the calculation load is less than the sigmoid and hyperbolic tangent functions has led to a higher preference for multi-layer networks, it is almost six times faster than the sigmoid and the hyperbolic tangent functions.

So what's the problem with RELU function?

- Because of this zero value region that gives us the speed of the process! So the learning is not happening in that area.

2.3.5.4 Leaky-Relu Function

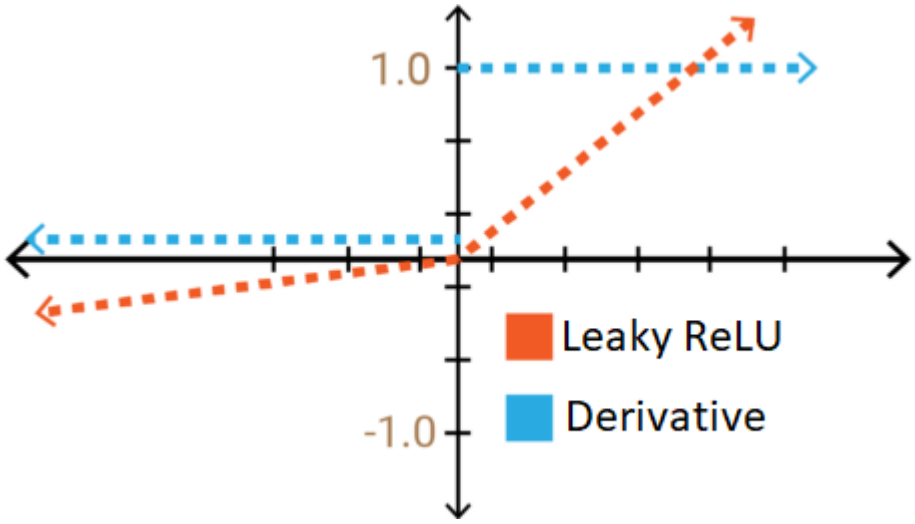


Figure 20

- This leaky value is given as a value of (0.01) if given a different value near zero, the name of the function changes randomly as Leaky ReLU. The definition range of the leaky-ReLU continues to be minus infinity. This is close to 0, but 0 with the value of the non-living gradients in the RELU lived in the negative region of learning to provide the values.

- The following table shows the mathematical equation of each function

ACTIVATION FUNCTION	EQUATION	RANGE
Linear Function	$f(x) = x$	$(-\infty, \infty)$
Step Function	$f(x) = \begin{cases} 0 \; for \; x < 0 \\ 1 \; for \; x \geq 0 \end{cases}$	$\{0, 1\}$
Sigmoid Function	$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$	$(0, 1)$
Hyperbolic Tanjant Function	$f(x) = \tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}$	$(-1, 1)$
ReLU	$f(x) = \begin{cases} 0 \; for \; x < 0 \\ x \; for \; x \geq 0 \end{cases}$	$[0, \infty)$
Leaky ReLU	$f(x) = \begin{cases} 0.01 \; for \; x < 0 \\ x \; for \; x \geq 0 \end{cases}$	$(-\infty, \infty)$

2.3.6 Layers in Convolutional Neural Networks.

2.3.6.1 Basic Overview On CNN.

- Convolutional Neural Networks are very similar to ordinary Neural Networks they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity
- Convolutional Network architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.
- Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a Convolutional Network have neurons arranged in 3 dimensions: **width, height, depth.**

The following figure illustrated the idea:

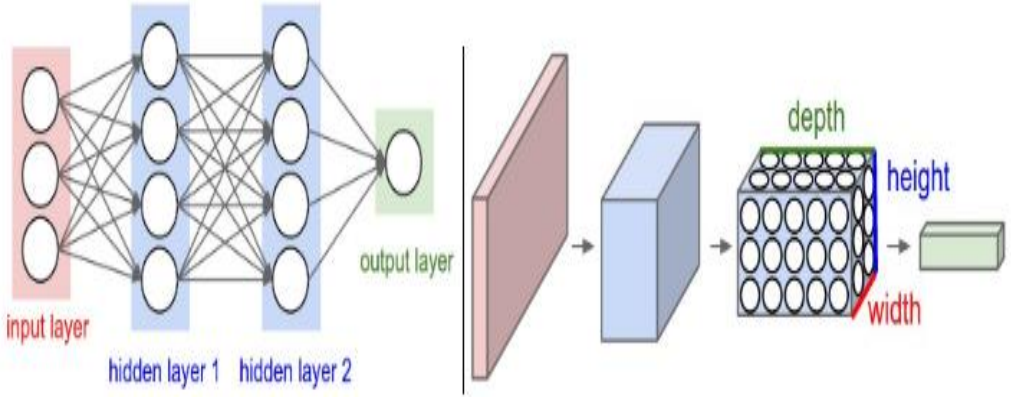


Figure 21

2.3.6.2 Layers Used to Build Convolutional Network

Left: A regular 3-layer Neural Network. Right: A Convolutional Network arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a Convolutional Network

transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels)

- As we described above, a simple Convolutional Network is a sequence of layers, and every layer of a Convolutional Network transform one volume of activations to another through a differentiable function.
- We use three main types of layers to build Convolutional Network architectures: Convolutional Layer, Pooling Layer, and Fully-Connected Layer (exactly as seen in regular Neural Networks). We will stack these layers to form a full Convolutional Network architecture.

Example Architecture:

We will go into more details below, but a simple Convolutional Network for CIFAR-10 classification could have the architecture [INPUT - CONV - RELU - POOL - FC]. In more detail:

- INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R, G, B.
- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters.
- RELU layer will apply an elementwise activation function, such as the max (0, x) max (0, x) thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]).
- POOL layer will perform a down sampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].
- FC (i.e. fully connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10.
- In this way, Convolutional Network transform the original image layer by layer from the original pixel values to the final class scores. Note that some layers contain parameters and other don't. In particular, the CONV/FC layers perform transformations that are a function of not only the activations in the input volume, but also of the parameters (the weights and biases of the neurons).
- On the other hand, the RELU/POOL layers will implement a fixed function. The parameters in the CONV/FC layers will be trained with gradient descent so that the class scores that the Convolutional

Network computations are consistent with the labels in the training set for each image.

In summary:

- A Convolutional Network architecture is in the simplest case a list of Layers that transform the image volume into an output volume (e.g. holding the class scores)
- There are a few distinct types of Layers (e.g. CONV/FC/RELU/POOL are by far the most popular)
- Each Layer accepts an input 3D volume and transforms it to an output 3D volume through a differentiable function
- Each Layer may or may not have parameters (e.g. CONV/FC do, RELU/POOL don't)
- Each Layer may or may not have additional hyper parameters (e.g. CONV/FC/POOL do, RELU doesn't)

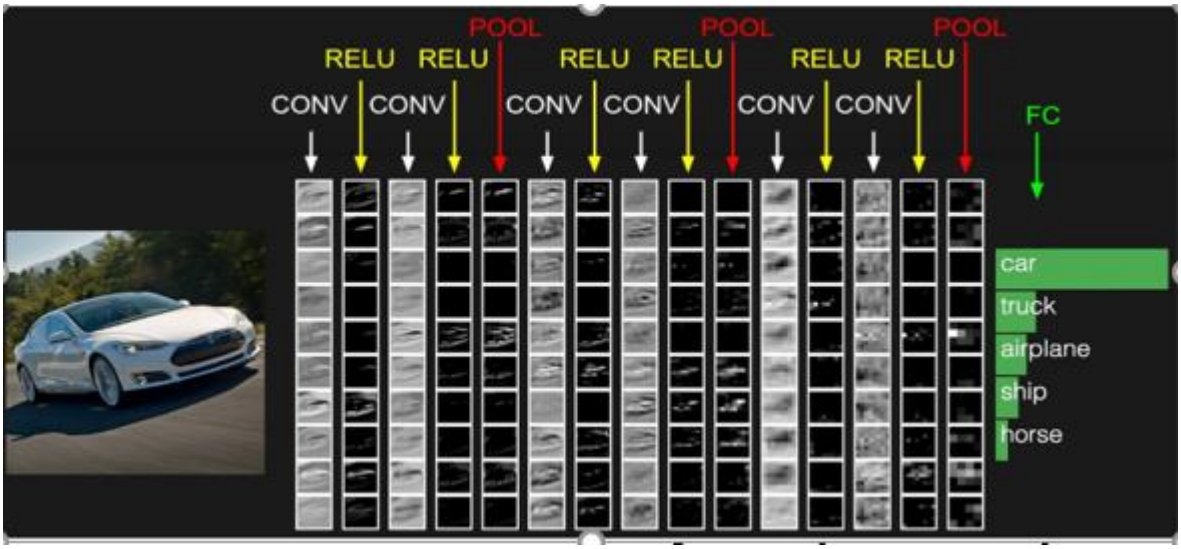
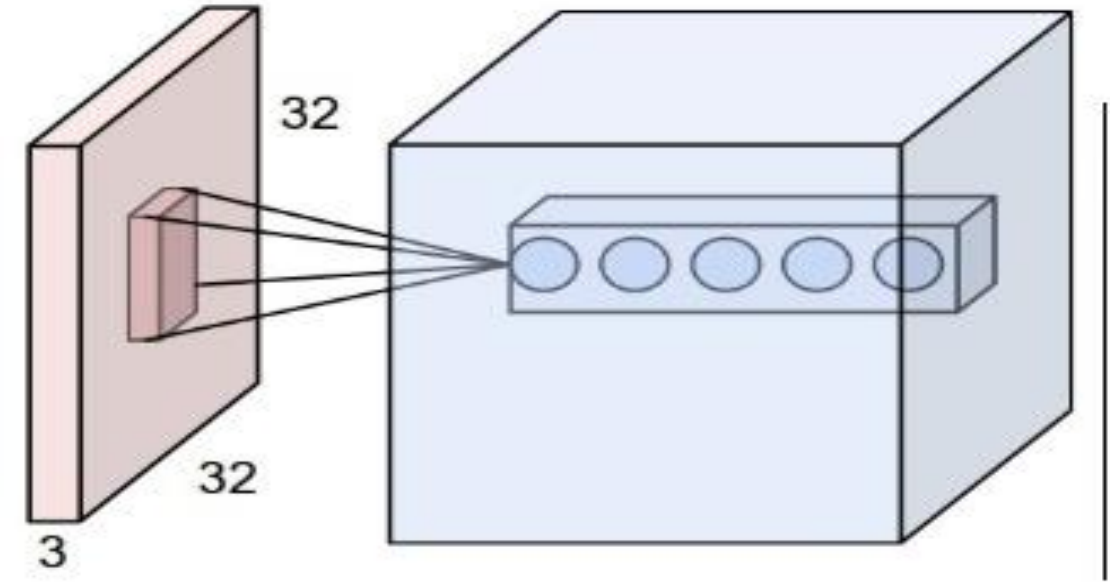


Figure 22

2.3.6.2.1 Convolutional Layer.

- The Convolutional Layer makes use of a set of learnable filters. A filter is used to detect the presence of specific features or patterns present in the original image (input). It is usually expressed as a matrix (MxMx3), with a smaller dimension but the same depth as the input file.
- This filter is convolved across the width and height of the input file, and a dot product is computed to give an activation map.
- Different filters which detect different features are convolved on the input file and a set of activation maps is outputted which is passed to the next layer in the CNN.

- Local Connectivity. When dealing with high-dimensional inputs such as images, as we saw above it is impractical to connect neurons to all neurons in the previous volume. Instead, we will connect each neuron to only a local region of the input volume. The spatial extent of this connectivity is a hyper parameter called the **receptive field** of the neuron (equivalently this is the filter size).



The following figures show the idea.

Figure 23

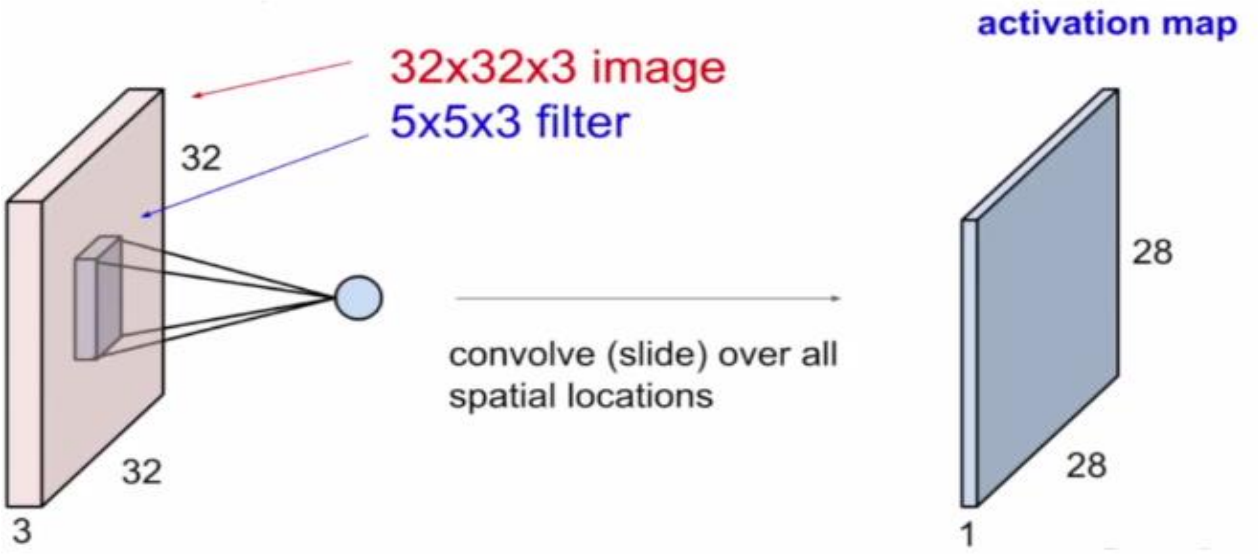


Figure 24

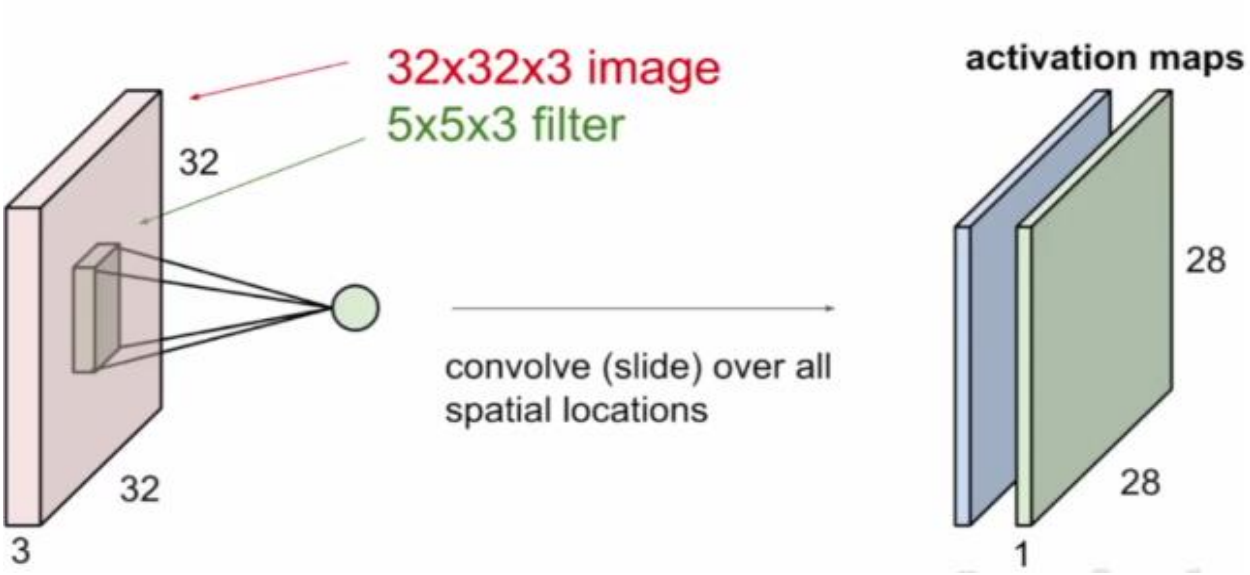


Figure 25

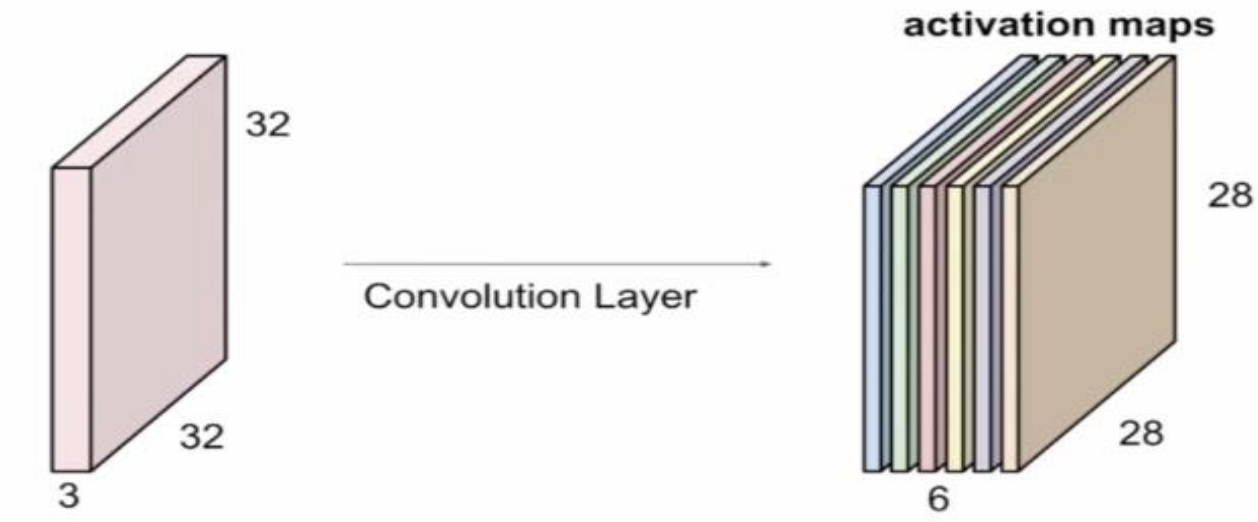


Figure 26

Three hyper parameters control the size of the output volume: the **depth, stride** and **zero-padding**

- First, the **depth** of the output volume is a hyper parameter: it corresponds to the number of filters we would like to use, each learning to look for something different in the input. As the number of filters used increases the depth of the resulting volume will increase.

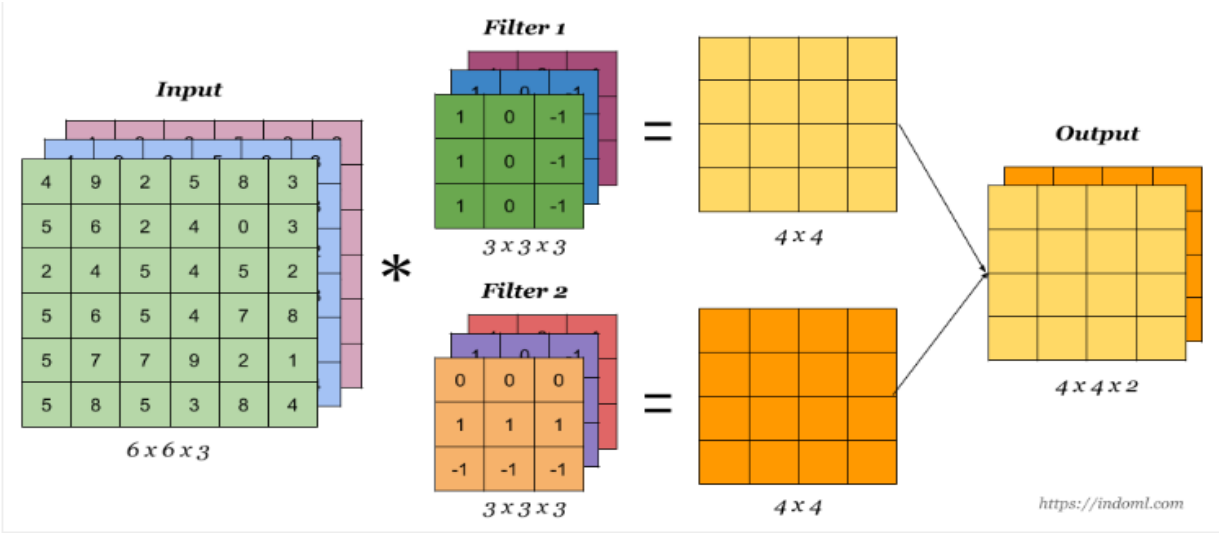


Figure 27

We notice that the number of channels of the output is the same as the number of filters used.

- Second, we must specify the **stride** with which we slide the filter. When the stride is 1 then we move the filters one pixel at a time, when the stride is 2 then the filters jump 2 pixels at a time as we slide them around. This will produce smaller output volumes spatially , as the stride, or movement, is increased, the resulting output will be smaller

Example for using a stride equals to 1

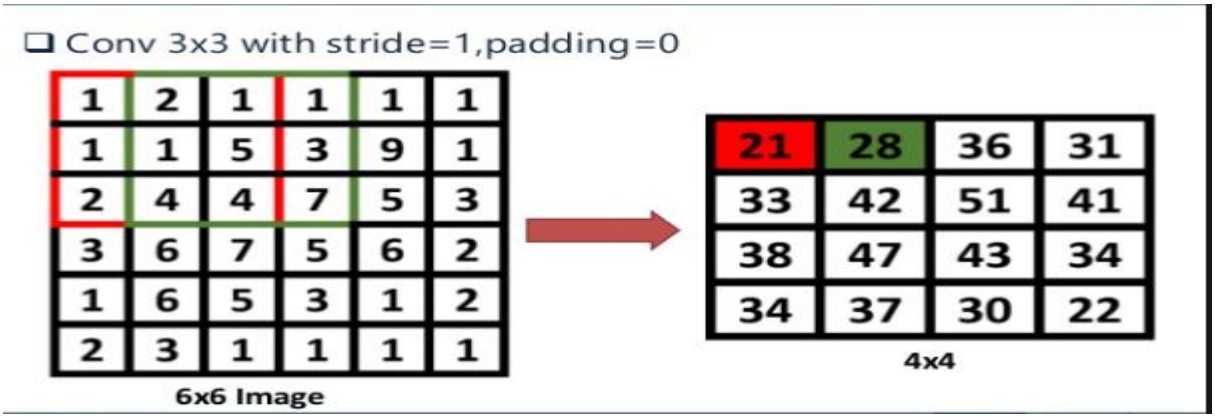


Figure 28

Example for using a stride equals to 2

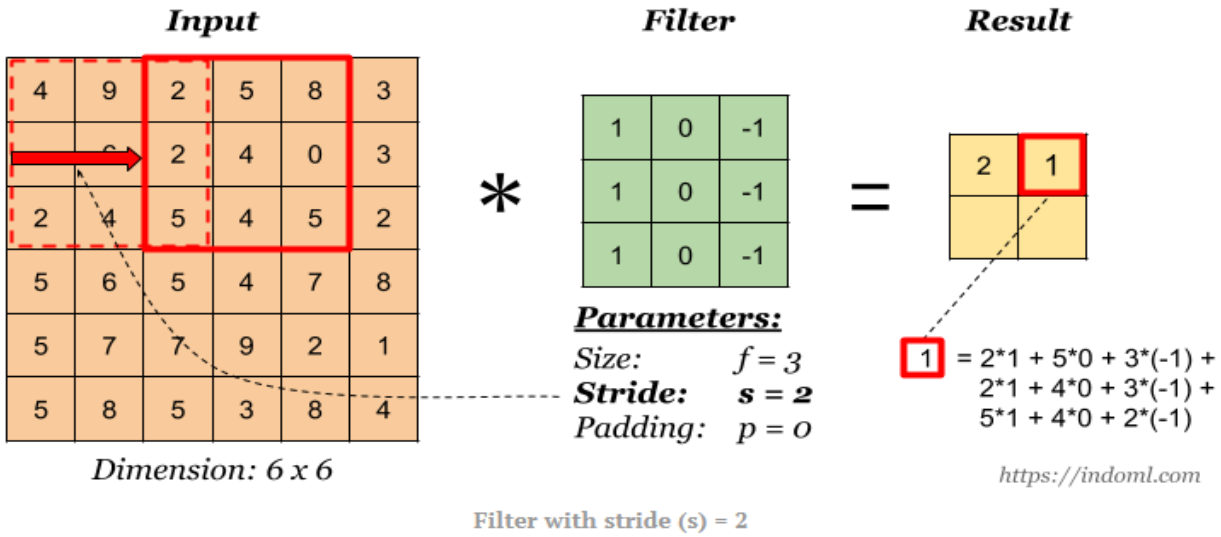


Figure 29

- Padding: The nice feature of zero padding is that it will allow us to control the spatial size of the output volume.

There is a formula which is used in determining the dimension of the activation maps:

$\frac{N+2P-F}{S} + 1$; where

- N = Dimension of image (input) file
  - P = Padding.
  - F = Dimension of filter.
  - S = Stride.

For example, for a 7x7 input and a 3x3 filter with stride 1 and pad 0 we would get a 5x5 output.

Important hints:

1) Padding may be used in order to avoid any information loss (because the convolution operation results in reducing the size of the original image) and hence the image may shrink.
2) By default, the stride equals 1 but in some cases the stride may be greater than 1 in order to speed up the processing time.
3) If the result of the equation $\frac{N+2P-F}{S} + 1$ is not an integer number, then we take the floor of that number.
4) There are 2 types of convolution:
   I. Valid CONV: in which the image is not padded.

II. Same CONV: in which we pad the original image such that the output has the same size as the input, in this case we use P = $\frac{F-1}{2}$ where 'F' is the size of the filter used.

Same CONV Example: using a filter of size 5x5x3 , input image of size 32x32x3 , hence we pad using P = $\frac{5-1}{2}$ = 2 so that the output image will also have a size of 32x32x3

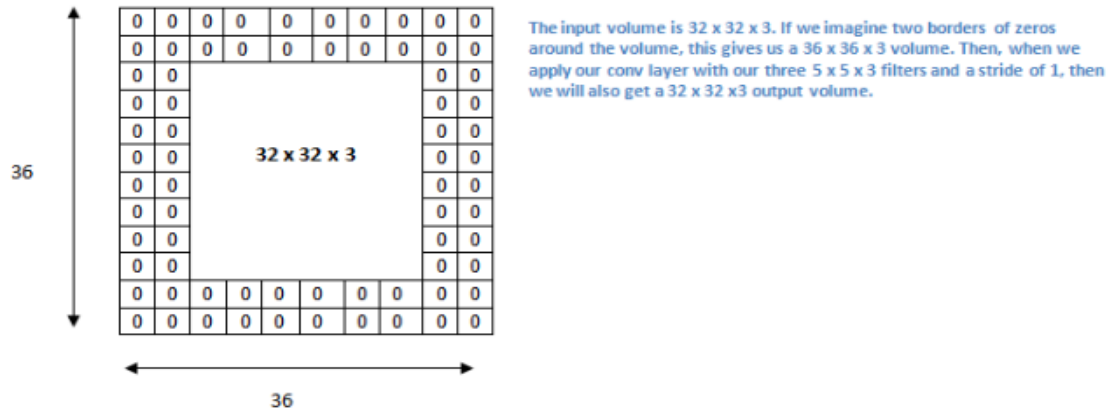


Figure 30

5) The filters used are usually of odd size, e.g.: 3x3, 5x5 7x7 and so on, each filter in every layer in the CNN network may be responsible for detecting a certain information in the input image as an example we can use sobel operators to detect the horizontal or vertical edges in our image.

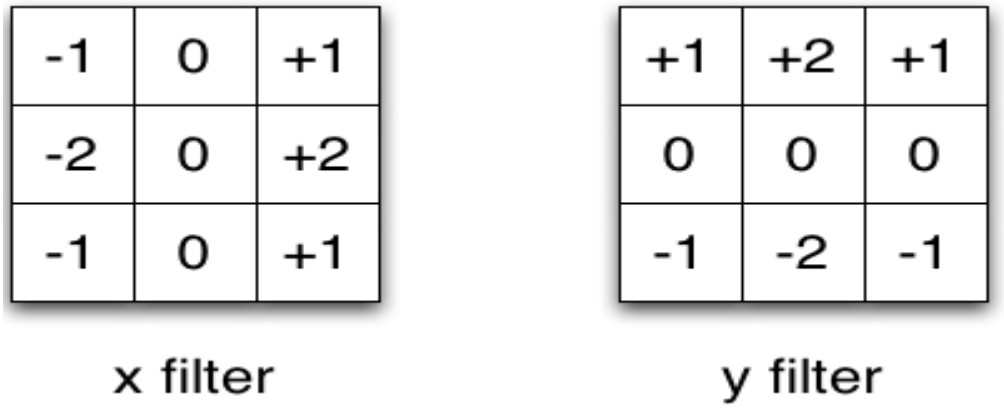


Figure 31

2.3.6.2.2 Polling Layer.

- It is common to periodically insert a Pooling layer in-between successive Conv layers in a Convolutional Network architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 down samples every depth slice in the input by 2 along both width and height.
- Max-pooling, like the name states; will take out only the maximum from a pool. This is done with the use of filters sliding through the input; and at every stride, the maximum parameter is taken out and the rest is dropped. This down samples the network.
- Unlike the convolution layer, the pooling layer does not alter the depth of the network, the depth dimension remains unchanged.

The following figure illustrates the max polling operation.

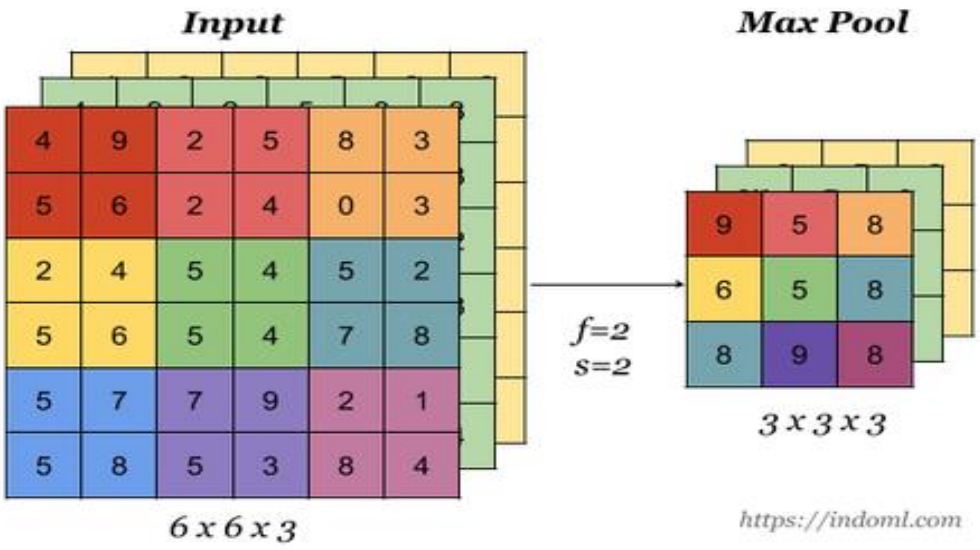


Figure 32

- Another type of polling layer is the average polling which is less commonly used than max polling on two-dimensional feature maps, pooling is typically applied in 2×2 patches of the feature map with a stride of 2. Average pooling involves calculating the average for each patch of the feature map. This means that each 2×2 square of the feature map is down sampled to the average value in the square.

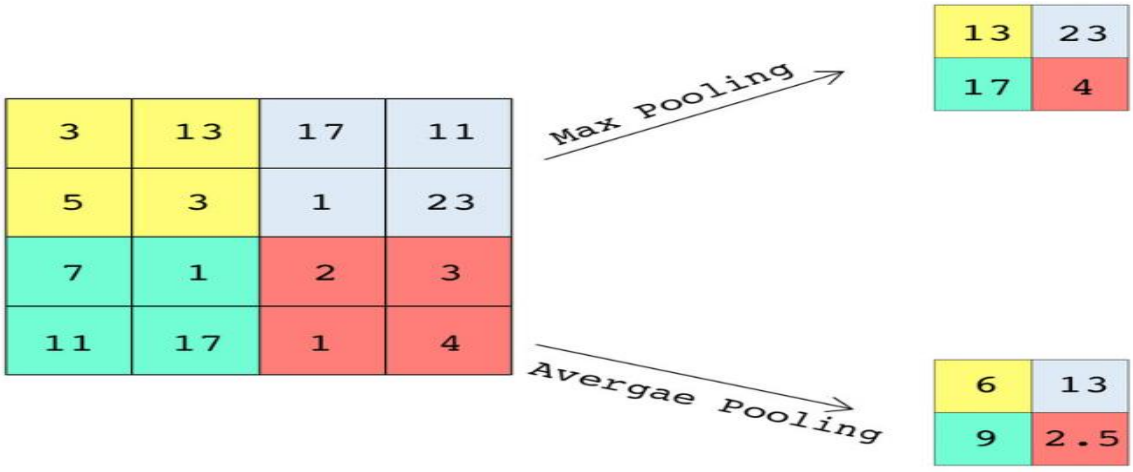


Figure 33

2.3.6.2.3 Fully Connected Layer.

- **The role of a fully connected layer in a CNN architecture:**

The objective of a fully connected layer is to take the results of the convolution/pooling process and use them to classify the image into a label (in a simple classification example).

The output of convolution/pooling is flattened into a single vector of values, each representing a probability that a certain feature belongs to a label. For example, if the image is of a cat, features representing things like whiskers or fur should have high probabilities for the label "cat", so in summary the polling and convolutional layers contain the main core of computations and the last few layers, which are typically the fully connected layers, put all propagated information into a single vector which is then used to make the final decision of the neural network regarding the input that was fed into the network at the input layer.

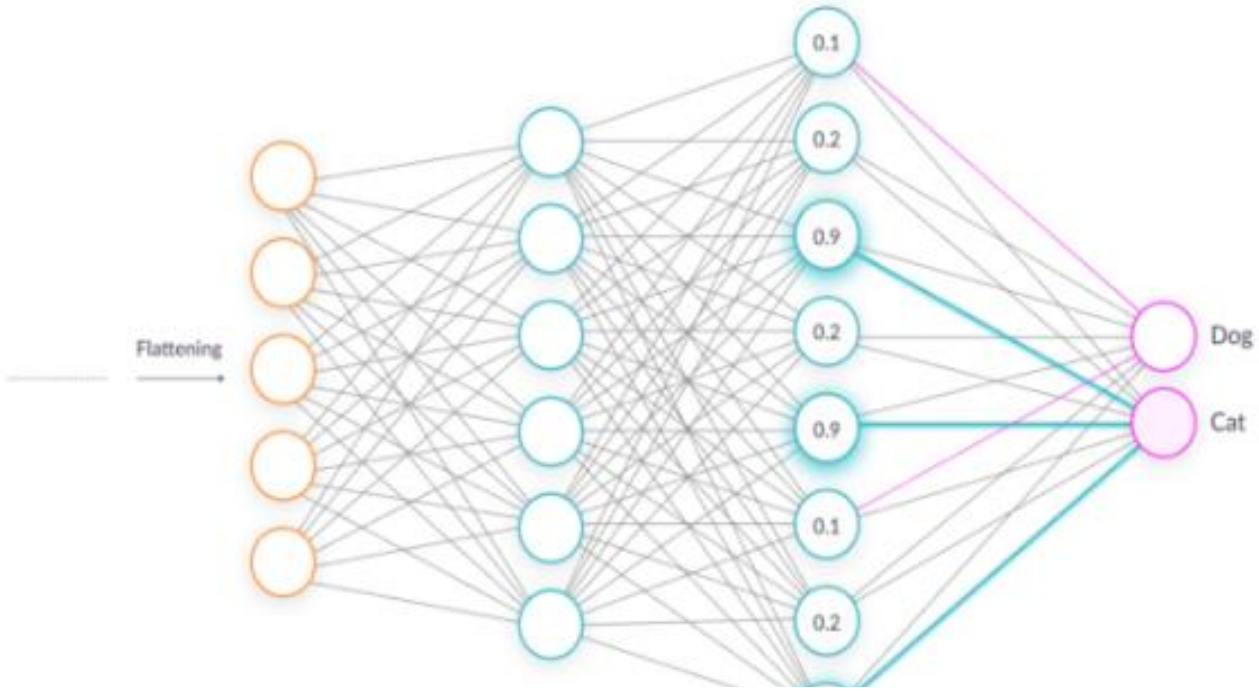


Figure 34

2.4 The Underfitting And Overfitting Problem

2.4.1 Introduction

Arguably, machine learning models have one sole purpose; to generalize well. Generalization is the model's ability to give sensible outputs to sets of input that it has never seen before.

A model learns relationships between the inputs, called features, and outputs, called labels, from a training dataset. During training the model is given both the features and the labels and learns how to map the former to the latter. A trained model is evaluated on a testing set, where we only give it the features and it makes predictions. We compare the predictions with the known labels for the testing set to calculate accuracy. Models can take many shapes, from simple linear regressions to deep neural networks, but all supervised models are based on the fundamental idea of learning relationships between inputs and outputs from training data.

Whenever working on a data set to predict or classify a problem, we tend to find accuracy by implementing a design model on first train set, then on test set. If the accuracy is satisfactory, we tend to increase accuracy of data-sets prediction either by increasing or decreasing data feature or features selection or applying feature engineering in our machine learning model. But sometime our model maybe giving poor result.

As shown in the figure, a model with too little capacity cannot learn the problem, whereas a model with too much capacity can learn it too well and overfit the training dataset. Both cases result in a model that does not generalize well.

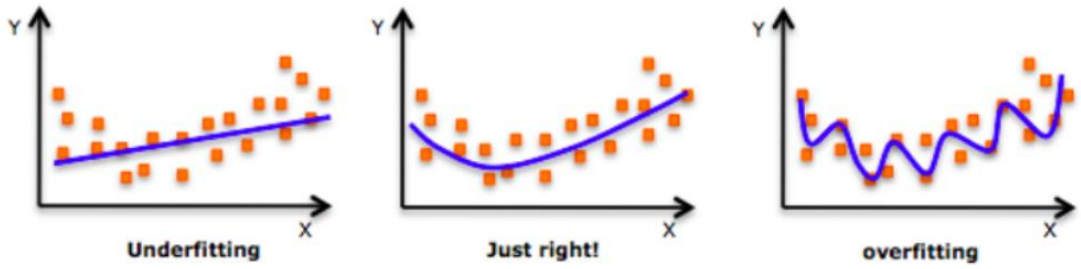


Figure 35

2.4.2 Terms of The Problem

Before explaining more in the underfitting and overfitting problem and how to solve them, we first need to clarify some terms that are essential for any machine learning model.

- Training Set: A training set is a group of sample inputs you can feed into the neural network in order to train the model. The neural network learns your inputs and finds weights for the neurons that can result in an accurate prediction.

- Training Error: The error is the difference between the known correct output for the inputs and the actual output of the neural network. During training, the training error is reduced until the model produces an accurate prediction for the training set.

- Validation Set: A validation set is another group of sample input which were not included in training and preferably are different from the samples in the training set. This is a “real life exercise” for the model: Can it generate correct predictions for an unknown set of inputs?

- Validation Error: The nice thing is that for the validation set, the correct outputs are already known, so it’s easy to compare the known correct prediction for the validation set with the actual model prediction, the difference between them is the validation error.

- Practically, when training a neural network model, you will attempt to gather a training set that is as large as possible and resembles the real population as much as possible. You will then break up the training set into at least two groups: one group will be used as the training set and the other as the validation set.

- Variance: high variance means that the model is not able to make accurate predictions on the validation set. The validation error will be large. Low variance means the model is successful in breaking out of its training data.

- Bias: high bias means the model is not “fitting” well on the training set. This means the training error will be large. Low bias means the model is fitting well, and training error will be low.

-

2.4.3 Model Building

Choosing a model can seem intimidating, but a good rule is to start simple and then build your way up. The simplest model is a linear regression, where the outputs are a linearly weighted combination of the inputs. In our model, we will use an extension of linear regression called polynomial regression to learn the relationship between x and y. Polynomial regression, where the inputs are raised to different powers, is still considered a form of “linear” regression even though the graph does not form a straight line The general equation for a polynomial is below.

(1.1)

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \cdots + \beta_n x^n + \varepsilon.$$

Here y represents the label and x is the feature. The beta terms are the model parameters which will be learned during training, and the epsilon is the error present in any model. Once the model has learned the beta values, we can plug in any value for x and get a corresponding prediction for y. A polynomial is defined by its order, which is the highest power of x in the equation. A straight line is a polynomial of degree 1 while a parabola has 2 degrees.

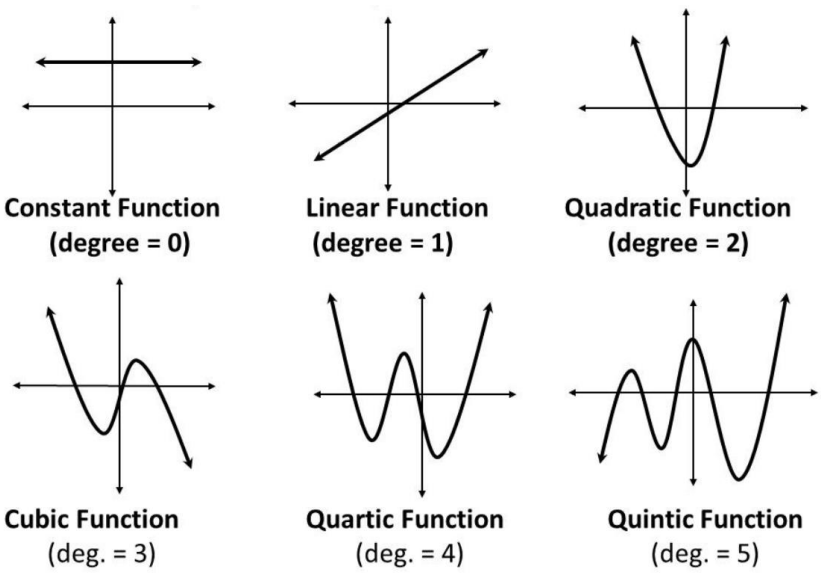


Figure 36: different degrees of polynomial function

-

2.4.4 Underfitting In Neural Networks

2.4.4.1 General Definition

The problem of Overfitting vs Underfitting finally appears when we talk about the polynomial degree. The degree represents how much flexibility is in the model, with a higher power allowing the model freedom to hit as many data points as possible. An underfit model will be less flexible and cannot account for the data.

This figure represents an underfit model with a 1-degree polynomial fit. As shown in the figure model function in orange is shown on top of the true function and the training observations.

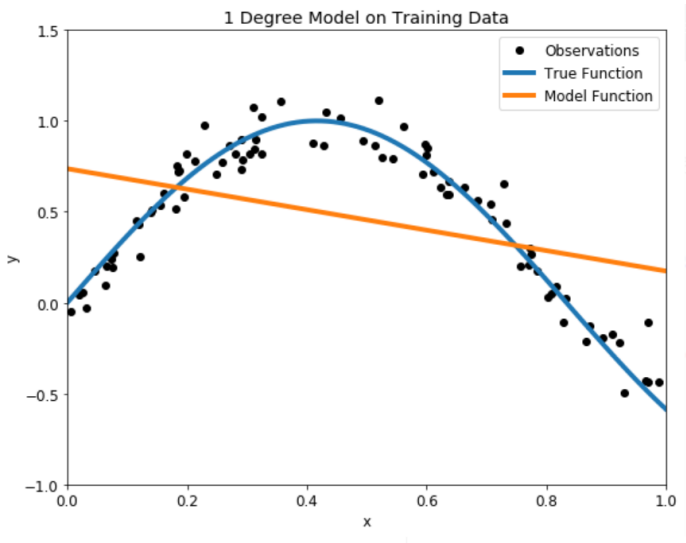


Figure 37:**Degree Model on training data**

Underfitting happens when the network is not able to generate accurate predictions on the training set, not to mention the validation set. The symptoms of underfitting are:

- High bias: poor predictions for the training set.
- High variance: poor predictions for the validation set

2.4.4.2 Methods to Avoid Underfitting.

We think that we should just increase the degree of the model to capture every change in the data but with a high degree of flexibility as shown in the figure, the model does its best to account for every single training point. While this would be acceptable if the training observations perfectly represented the true function, because there is noise in the data, our model ends up fitting the noise. This is a model with a high variance, because it will change significantly depending on the training data. The predictions on the test set are better than the one-degree model, but the twenty-five-degree model still does not learn the relationship because it essentially memorizes the training data and the noise.

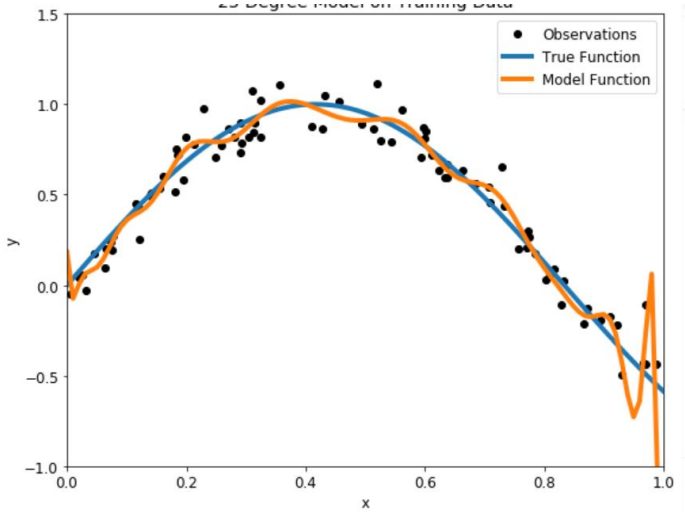


Figure 38:**Degree model on training data**

Our problem is that we want a model that does not memorize the training data, but learns the actual relationship, and here comes the validation method.

A basic approach would be to use a validation set in addition to the training and testing set. This presents a few problems though: we could just end up overfitting to the validation set and we would have less training data. A smarter implementation of the validation concept is k-fold cross-validation.

The idea is straightforward: rather than using a separate validation set, we split the training set into several subsets, called folds. Let's use five folds as an example. We perform a series of train and evaluate cycles where each time we train on 4 of the folds and test on the 5th, called the hold-out set. We repeat this cycle 5 times, each time using a different fold for evaluation. At the end, we average the scores for each of the folds to determine the overall performance of a given model. This allows us to optimize the model before deployment without having to use additional data.

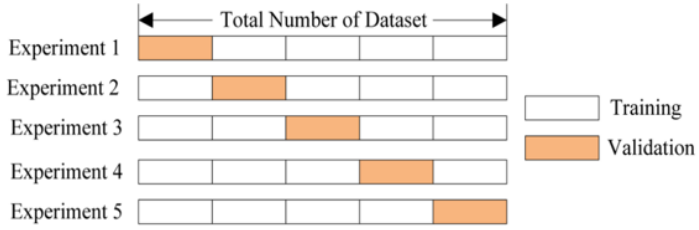


Figure 39:**five-fold cross validation**

For our problem, we can use cross-validation to select the best model by creating models with a range of different degrees and evaluate each one using 5-fold cross-validation. The model with the lowest cross-validation score will perform best on the testing data and will achieve a balance between underfitting and overfitting. I choose to use models with degrees from 1 to 40 to cover a wide range. To compare models, we compute the mean-squared error, the average distance between the prediction and the real value squared. The following table shows the cross-validation results ordered by lowest error and the graph shows all the results with error on the y-axis.

	degrees	cross_valid
0	4	0.010549
1	5	0.010637
2	7	0.010665
3	6	0.010887
4	8	0.011182
5	3	0.011695
6	9	0.011757
7	11	0.011769
8	10	0.011902
9		

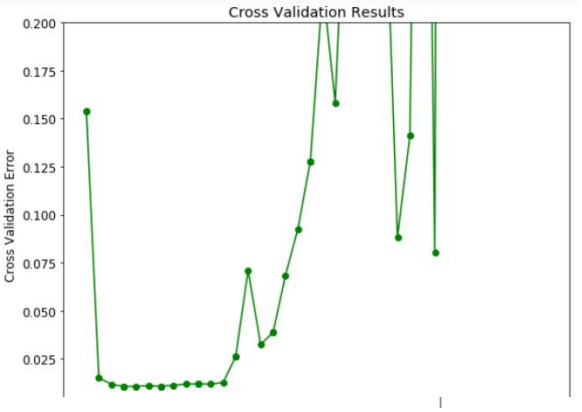


Figure 40:**Cross validation results**

The cross-validation error with the underfit and overfit models is off the chart! A model with 4 degrees appears to be optimal. To test out the results, we can make a 4-degree model and view the training and testing predictions.

Generally, using a linear model for image recognition will generally result in an underfitting model. Alternatively, when experiencing underfitting in your deep neural network this is probably caused by dropout. Dropout randomly sets activations to zero during the training process to avoid overfitting. This does not happen during prediction on the validation/test set. If this is the case, you can remove dropout. If the model is now massively overfitting you can start adding dropout in small pieces.

2.4.5 Overfitting In Neural Networks

2.4.5.1 General Definition

Overfitting happens when the neural network is very good at learning its training set but cannot generalize beyond the training set (known as the generalization problem), the symptoms of overfitting are:

- Low bias : accurate predictions for the training set.
- High variance: poor ability to generate predictions for the validation set.

2.4.5.2 Methods To Solve And Avoid Overfitting

The following are common methods used to improve the generalization of a neural network, providing better predictive performance when applied to the unknown validation samples. The goal is to improve variance.

- Add more data: The first step is of course to collect more data. However, in most cases you will not be able to. Let's assume you have collected all the data.
- Data augmentation: It includes things as shown in the figure like randomly rotating the image, zooming in, adding a color filter etc. Data augmentation only happens to the training set and not on the validation/test set. It can be useful to check if you are using too much data augmentation. For example, if you zoom in so much that features of a cat are not visible anymore, then the model is not going to get better from training on these images.

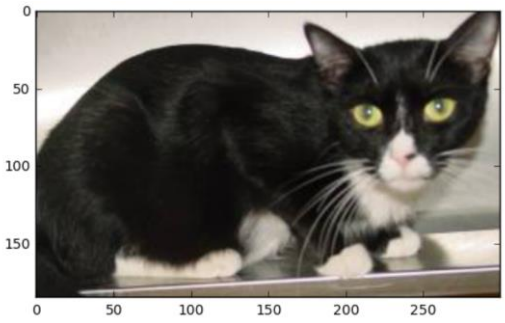


Figure 41:**Original image**

Figure 42:**Augmented image**

- Add regularization: The three most popular options are dropout, L1 regularization and L2 regularization. In deep learning you will mostly see dropout. Dropout deletes a random sample of the activations (makes them zero) in training. It can be applied in the fully connected layers at the end of the model and it can also be applied to the convolutional layers. Be aware that dropout causes information to get lost. If you lose something in the first layer, it gets lost for the whole network. Therefore, a good practice is to start with a low dropout in the first layer and then gradually increase it. Regularization is a slightly more complex technique which involves modifying the error function (usually calculated as the sum of squares on the errors for the individual training or validation samples)Without getting into the math, the trick is to add a term to the error function, which is intended to decrease the weights and biases, smoothing outputs and making the network less likely to overfit.

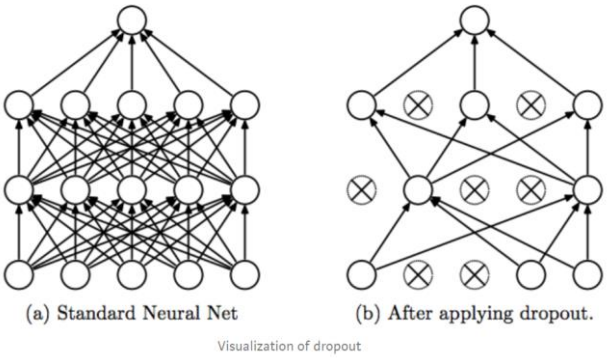


Figure 43:**Visualization of dropout**

- Early Stopping: Early stopping is a kind of cross-validation strategy where we keep one part of the training set as the validation set. When we see that the performance on the validation set is getting worse, we immediately stop the training on the model. In the figure below, we will stop training at the dotted line since after that our model will start overfitting on the training data.

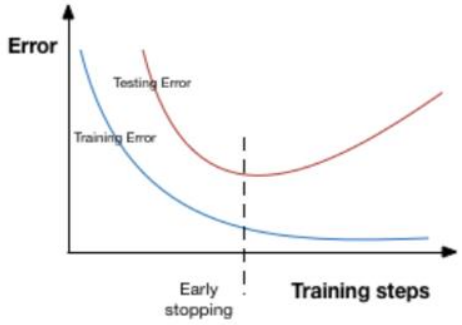


Figure 44:**training steps vs error**.

2.5 Tools to Use to Build A Neural Networks

2.5.1 Introduction

We know in the software industry that without an interface, libraries, and organized tools, software development proves a nightmare. When we combine these essentials, it becomes a framework or platform for easy, quick, and meaningful software development.

ML frameworks help ML developers to define ML models in precise, transparent, and concise ways. ML frameworks used to provide pre-built and optimize components to help in model building and other tasks. There are famous frameworks commonly used as TensorFlow and PyTorch, we expected to use both, so we wanted to know the differences between them.

TensorFlow: TensorFlow is an open source deep learning framework created by developers at Google and released in 2015, It supports regressions, classifications, and neural networks like complicated tasks and algorithms. You can run it on CPUs & GPUs both. TensorFlow creates a static graph, you first must define the entire computation graph of the model and then run your ML model. It’s more difficult than PyTorch to learn but it has larger community and easier to find solutions to your problem as it’s older than PyTorch.

PyTorch: PyTorch is based on Torch and has been developed by Facebook, it's used by Facebook, IBM, Yandex, and Idiap Research Institute. PyTorch believes in a dynamic graph, you can define/manipulate your graph on-the-go. This is particularly helpful while using variable length inputs in RNNs. Torch is flexible and offers high-end efficiencies and speed. It offers a lot of pre-trained modules. It's easier to learn but it is new compared to TensorFlow, so it has smaller community than TensorFlow.

2.5.2 Building A CNN Model

In this section we will introduce our implementation of building a CNN, the tools we used, the problems we faced and how we solved them.

Tools used :

- IDE: PyCharm
- Python version: 3.6.2
- Modules versions

- TensorFlow:
  - Tensorflow == 2.0.0
  - Tensorflow-estimator == 2.0.0
  - Tensorflow-gpu == 1.14.0
  - Tensorflow-tensorboard == 1.5.1
- Numpy:
  - Numpy == 1.16.4
- Matlabplotlib.pyplot:
  - Matplotlib == 3.1.1

Code :

```
import TensorFlow as tf

from tensorboard.plugins.hparams.api import KerasCallback

from tensorflow import keras

import numpy as np

import matplotlib.pyplot as plot
```

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']


if __name__ == '__main__':
    data = keras.datasets.fashion_mnist
    (train_images, train_labels), (test_images, test_labels) = data.load_data()
    train_images = train_images / 255.0
    test_images = test_images / 255.0

    # lets do our architecture
    # sequential is we put our layers in sequence
    model = keras.Sequential([
        keras.layers.Flatten(input_shape=(28, 28)),  # flatten the input to be a list
        # dense means a fully connected layer which is our hidden layer here relu is fast
        keras.layers.Dense(128, "relu"),
        keras.layers.Dense(64, "relu"),
        keras.layers.Dropout(0.25),


        # this is our output layer and each one of them represent what item is it ,softmax is used so that all the
        # neurons in this layer add up to 1 making it a probability
        keras.layers.Dense(10, "softmax")
```

```
    ])

    # adam and loss function here are standard we may look at these later and how do they work

    # accuracy this is what we want to measure how accurate is our network

    # compile to give the model some parameters

    model.compile(optimizer="adam", loss="sparse_categorical_crossentropy", metrics=["accuracy"])


    # this is used to train our network check what epochs is in the copybook

    model.fit(train_images, train_labels, epochs=100)


    # now its time to check how our network works so we will see the accuracy

    test_loss, test_acc = model.evaluate(test_images, test_labels)

    print("our accuracy is ", test_acc)


# prediction = model.predict(test_images)

# print(prediction[0])
```

Dataset:

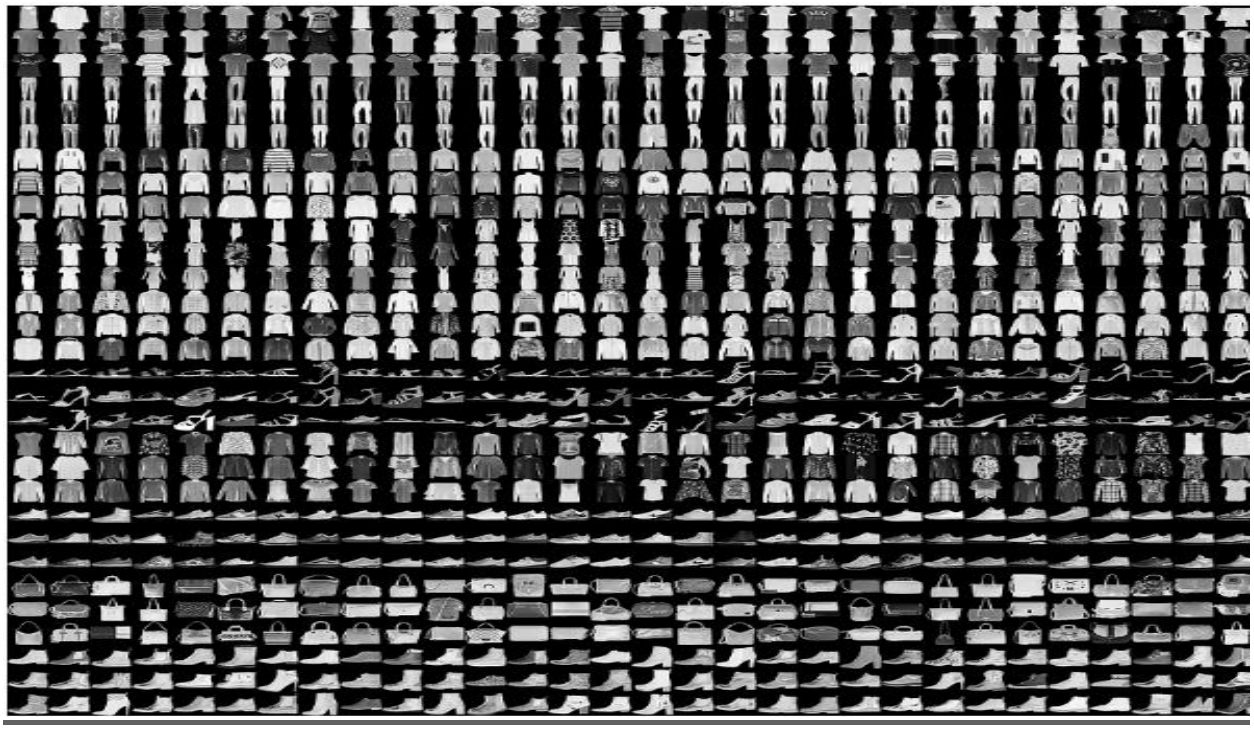


Figure 45

Lables :

Label	Class
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Figure 46

Then we tried running the code with different numbers of epochs to get the best accuracy and here are the results :

No. of epochs	Accuracy after training	Accuracy of test dataset

10	91.25%	88.8%
100	98%	88.88%

We are faced here with a strong example of overfitting as the accuracy of our neural network was 0.9125 and the accuracy of our test dataset is 0.888 when using 10 epochs while when using 100 epochs the accuracy of our neural network increased to 0.98 while the accuracy when using the test dataset was only increased to 0.8888 so we will try to do a dropout it may solve this problem.

Other architectures results:

Layers	Accuracy after training	Accuracy of test data	Number of epochs	Simulation time/min	Dropout
1 Hidden layer of 128 neurons	87%	86%	5	1.5	0
1 Hidden layer of 128 neurons	90%	88%	10	3	0
1 Hidden layer of 128 neurons	98%	90%	100	15	0.25
2 hidden layers of sizes 128 and 64	86%	85%	5	2	0
2 hidden layers of sizes 128 and 64	89%	87%	10	5	0
2 hidden layers of	97%	89.5%	100	20	0.25

sizes 128 and 64					

2.5.3 Autonomous Driving Dataset

For our implementation we use two data sets one is called udacity data set and the other is called VOC data set 's annotation file for the udacity data set it has the CSV format while the VOC data set has an XML annotation file format And the following figures show some examples of the data sets used


3.1 Yolo Algorithm (You-Only-Look-Once)

The idea is to divide the image into multiple grids. Then we change the label of our data such that we implement both localization and classification algorithm for each grid cell. Let me explain this to you with one more infographic.

We start with placing a grid on top of the input image. Then, for each of the grid cells, we run the classification and localization algorithm we saw at the beginning of the blog. The labels for training, for each grid cell, will be like what we saw earlier, with an 8-dimensional output vector:

$$Y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

Figure 47:**yolo vector**

For each cell, we will get a result whether there is an object or not. For example:

$$\text{Object of class 2 (i.e. car) detected: } \begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad ; \quad \text{No object detected: } \begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix}$$

Figure 48:yolo vector evaluated

The object is "assigned" to the specific cell looking to where the center falls.

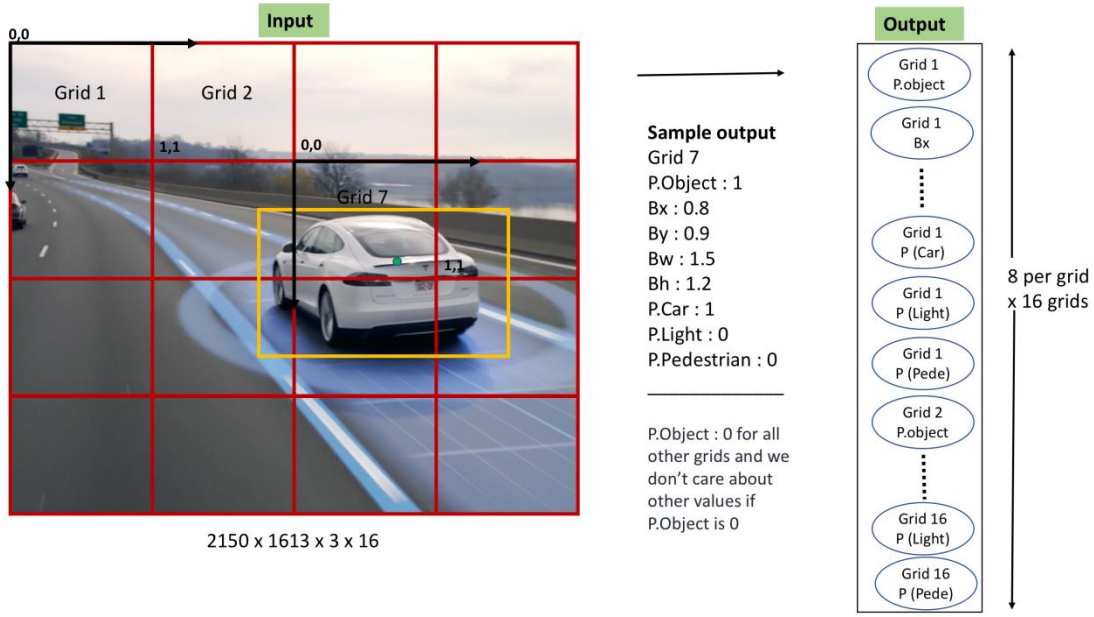
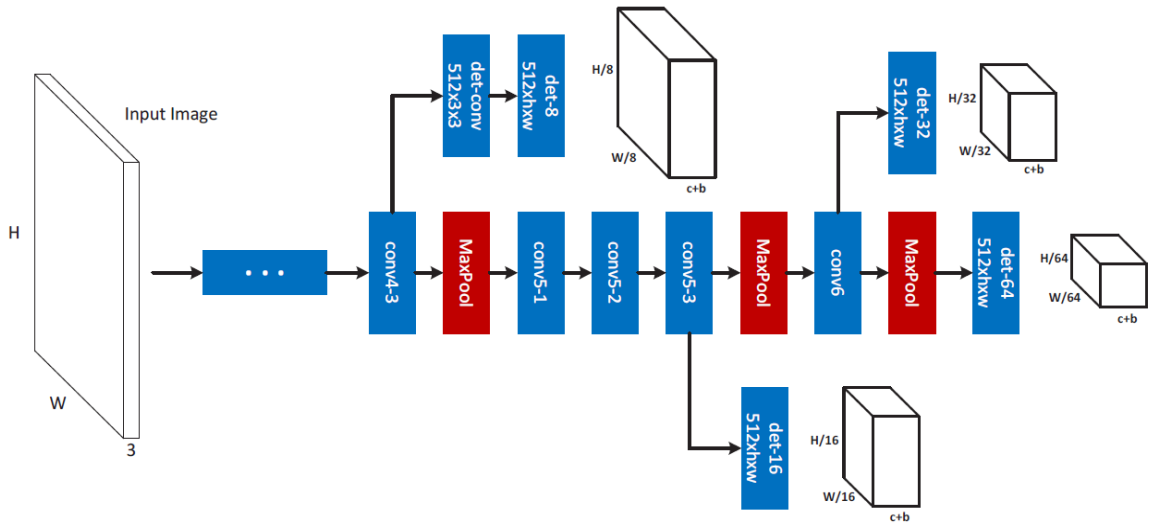


Figure 49:yolo window

3.2 Multi scale CNN (MS-CNN)

A These have this network consists of two sub-networks: an object proposal network and an accurate detection network. Both are learned end-to-end and share computations. However, to ease the inconsistency between the sizes of objects and receptive fields, object detection is performed with multiple output layers, each focusing on objects within certain scale ranges The intuition is that lower network layers, such as "conv-3," have smaller receptive fields, better matched to detect small objects. Conversely, higher layers,

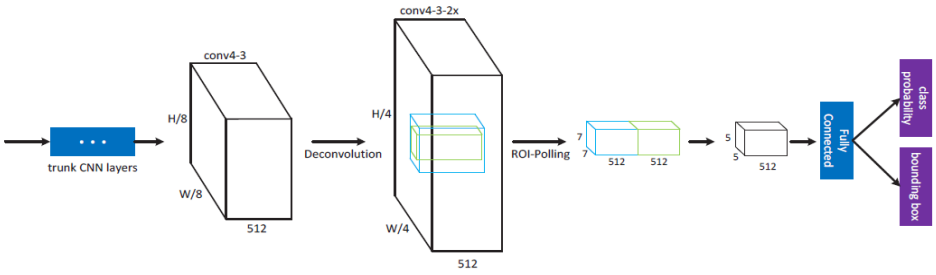
such as “conv-5,” are best suited for the detection of large objects. The complimentary detectors at different output layers are combined to form a strong multi-scale detector



Proposal sub-network of the MS-CNN. The bold cubes are the output tensors of the network. $h \times w$ is the filter size, c the number of classes, and b the number of bounding box coordinates.

Figure 50:architecture overview

The detailed architecture of the MS-CNN proposal network is shown in this Figure The network detects objects through several detection branches. The results by all detection branches are simply declared as the final proposal detections. The network has a standard CNN trunk, depicted in the center of the figure, and a set of output branches, which emanate from different layers of the trunk. These branches consist of a single detection layer.



Object detection sub-network of the MS-CNN. “trunk CNN layers” are shared with proposal sub-network. W and H are the width and height of the input image. The green (blue) cubes represent object (context) region pooling. “class probability” and “bounding box” are the outputs of the detection sub-network.

Figure 51:Detection layer

Detection layer Although the proposal network could work as a detector itself, it is not strong, since its sliding windows do not cover objects well. To increase detection accuracy, a detection network is added. Following ,a ROI pooling layer is first used to extract features of a fixed dimension (e.g. 7×7×512). The features are then fed to a fully connected layer and output layers, as shown in Fig. 4. A deconvolution layer, described in Section 4.1, is added to double the resolution of the feature maps.

The parameters and configuration with results of training

Table 1. Parameter configurations of the different models.

		det-8		det-16		det-32		det-64	ROI	FC
car	filter	5x5	7x7	5x5	7x7	5x5	7x7	5x5	7x7	4096
	anchor	40x40	56x56	80x80	112x112	160x160	224x224	320x320		
ped/cyc	filter	5x3	7x5	5x3	7x5	5x3	7x5	5x3	7x5	2048
	anchor	40x28	56x36	80x56	112x72	160x112	224x144	320x224		
caltech	filter	5x3	7x5	5x3	7x5	5x3	7x5	5x3	8x4	2048
	anchor	40x20	56x28	80x40	112x56	160x80	224x112	320x160		

Table 2. Detection recall of the various detection layers on KITTI validation set (car), as a function of object hight in pixels.

	det-8	det-16	det-32	det-64	combined
25≤height<50	0.9180	0.3071	0.0003	0	0.9360
50≤height<100	0.5934	0.9660	0.4252	0	0.9814
100≤height<200	0.0007	0.5997	0.9929	0.4582	0.9964
height≥200	0	0	0.9583	0.9792	0.9583
all scales	0.6486	0.5654	0.3149	0.0863	0.9611

Figure 52:MSCNN result

Table 4. Results on the KITTI benchmark test set (only published works shown).

Method	Time	Cars			Pedestrians			Cyclists		
		Easy	Mod	Hard	Easy	Mod	Hard	Easy	Mod	Hard
LSVM-MDPM-sv [35]	10s	68.02	56.48	44.18	47.74	39.36	35.95	35.04	27.50	26.21
DPM-VOC-VP [36]	8s	74.95	64.71	48.76	59.48	44.86	40.37	42.43	31.08	28.23
SubCat [16]	0.7s	84.14	75.46	59.71	54.67	42.34	37.95	-	-	-
3DVP [37]	40s	87.46	75.77	65.38	-	-	-	-	-	-
AOG [38]	3s	84.80	75.94	60.70	-	-	-	-	-	-
Faster-RCNN [9]	2s	86.71	81.84	71.12	78.86	65.90	61.18	72.26	63.35	55.90
CompACT-Deep [15]	1s	-	-	-	70.69	58.74	52.71	-	-	-
DeepParts [39]	1s	-	-	-	70.49	58.67	52.78	-	-	-
FilteredICF [40]	2s	-	-	-	67.65	56.75	51.12	-	-	-
pAUCEnsT [41]	60s	-	-	-	65.26	54.49	48.60	51.62	38.03	33.38
Regionlets [20]	1s	84.75	76.45	59.70	73.14	61.15	55.21	70.41	58.72	51.83
3DOP [5]	3s	**93.04**	88.64	**79.10**	81.78	67.47	64.70	78.39	68.94	61.37
SDP+RPN [42]	0.4s	90.14	88.85	78.38	80.09	70.16	64.82	81.37	73.74	65.31
MS-CNN	0.4s	90.03	**89.02**	76.11	**83.92**	**73.70**	**68.31**	**84.06**	**75.46**	**66.07**

Figure 53:compared results

3.3 Single Shot Detector MultiBox

These This time, SSD (Single Shot Detector) is reviewed. By using SSD, we only need to take one single shot to detect multiple objects within the image, while regional proposal network (RPN) based approaches such as R-CNN series that need two shots, one for generating region proposals, one for detecting the object of each proposal. Thus, SSD is much faster compared with two-shot RPN-based approaches

After going through a certain of convolutions for feature extraction, we obtain a feature layer of size m×n (number of locations) with p channels, such as 8×8 or 4×4 above. And a 3×3 conv is applied on this m×n×p feature layer.

For each location, we got k bounding boxes. These k bounding boxes have different sizes and aspect ratios. The concept is, maybe a vertical rectangle is more fit for human, and a horizontal rectangle is more fit for car.

For each of the bounding box, we will compute c class scores and 4 offsets relative to the original default bounding box shape.

Thus, we got (c+4)kmn outputs.

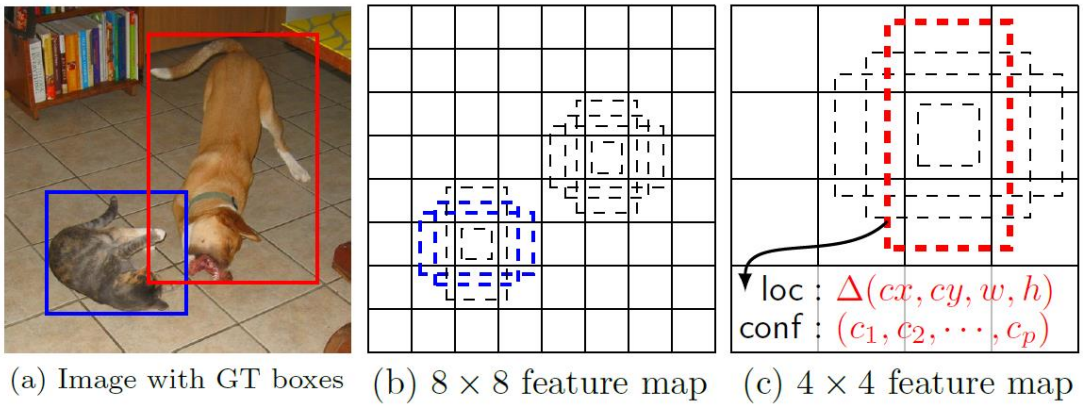


Figure 54:SSD example

Default boundary boxes are chosen manually. SSD defines a scale value for each feature map layer. Starting from the left, Conv4_3 detects objects at the smallest scale 0.2 (or 0.1 sometimes) and then increases linearly to the rightmost layer at a scale of 0.9. Combining the scale value with the target aspect ratios, we compute the width and the height of the default boxes. For layers making 6 predictions

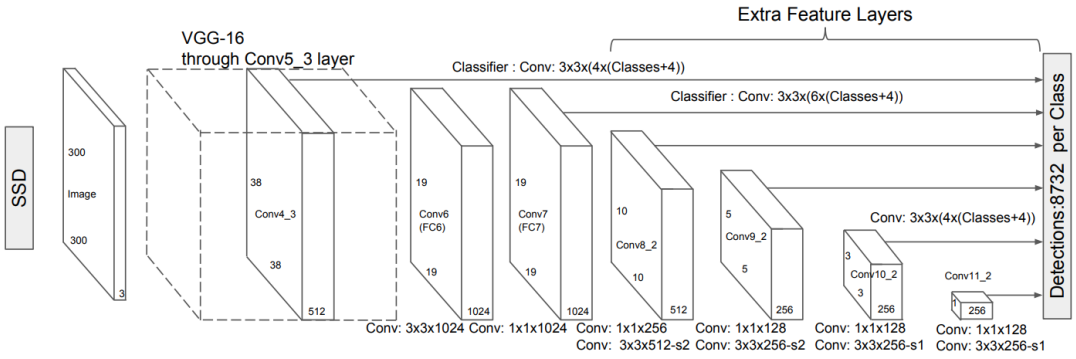


Figure 55:SSD overview

3.4. Understanding Region-Based Convolutional Neural Network

Object detection is the process of finding and classifying objects in an image. One deep learning approach, regions with convolutional neural networks (R-CNN), combines rectangular region proposals with convolutional neural network features. R-CNN is a two-stage detection algorithm. The first stage identifies a subset of regions in an image that might contain an object. The second stage classifies the object in each region. Applications for R-CNN object detectors include: Autonomous driving, Smart surveillance systems, Facial recognition, Models for object detection using regions with CNNs are based on the following three processes: Find regions in the image that might contain an object. These regions are called region proposals.

Extract CNN features from the region proposals,Classify the objects using the extracted features. There are three variants of an R-CNN. Each variant attempts to optimize, speed up, or enhance the results of one or more of these processes.

3.4.1 Intuition of RCNN

Instead of working on a massive number of regions, the RCNN algorithm proposes a bunch of boxes in the image and checks if any of these boxes contain any object. RCNN uses selective search to extract these boxes from an image (these boxes are called regions). Let's first understand what selective search is and how it identifies the different regions. There are basically four regions that form an object: varying scales, colors, textures, and enclosure. Selective search identifies these patterns in the image and based on that, proposes various regions. Here is a brief overview of how selective search works:

It first takes an image as input, Then, it generates initial sub-segmentations so that we have multiple regions from this image, The technique then combines the similar regions to form a larger region (based on color similarity, texture similarity, size similarity, and shape compatibility):
Finally, these regions then produce the final object locations (Region of Interest),Below is a succinct summary of the steps followed in RCNN to detect objects, We first take a pre-trained convolutional neural network, Then, this model is retrained. We train the last layer of the network based on the number of classes that need to be detected. The third step is to get the Region of Interest for each image. We then reshape all these regions so that they can match the CNN input size, after getting the regions, we train SVM to classify objects and background. For each class, we train one binary SVM,Finally, we train a linear regression model to generate tighter bounding boxes for each identified object in the image.

3.4.2 Problems with RCNN

So far, we've seen how RCNN can be helpful for object detection. But this technique comes with its own limitations. Training an RCNN model is expensive and slow thanks to the below steps:

Extracting 2,000 regions for each image based on selective search. Extracting features using CNN for every image region. Suppose we have N images, then the number of CNN features will be N*2,000. The entire process of object detection using RCNN has three models: CNN for feature extraction. Linear SVM classifier for identifying objects Regression model for tightening the bounding boxes. All these processes combine to make RCNN very slow. It takes around 40-50 seconds to make predictions for each new image, which essentially makes the model cumbersome and practically impossible to build when faced with a gigantic dataset. Here's the good news – we have another object detection technique which fixes most of the limitations we saw in RCNN.

3.5 Understanding Fast RCNN

3.5.1 Intuition of Fast RCNN

What else can we do to reduce the computation time a RCNN algorithm typically takes? Instead of running a CNN 2,000 times per image, we can run it just once per image and get all the regions of interest (regions containing some object). Ross Girshick, the author of RCNN, came up with this idea of running the CNN just once per image and then finding a way to share that computation across the 2,000 regions. In Fast RCNN, we feed the input image to the CNN, which in turn generates the convolutional feature maps. Using these maps, the regions of proposals are extracted. We then use a RoI pooling layer to reshape all the proposed regions into a fixed size, so that it can be fed into a fully connected network. Let's break this down into steps to simplify the concept: As with the earlier two techniques, we take an image as an input. This image is passed to a ConvNet which in turns generates the Regions of Interest.

A RoI pooling layer is applied on all of these regions to reshape them as per the input of the ConvNet. Then, each region is passed on to a fully connected network.A SoftMax layer is used on top of the fully connected network to output classes. Along with the SoftMax layer, a linear regression layer is also used parallelly to output bounding box coordinates for predicted classes. So, instead of using three different models (like in RCNN), Fast RCNN uses a single model which extracts features from the regions, divides them into different classes, and returns the boundary boxes for the identified classes simultaneously. To break this down even further, I'll visualize each step to add a practical angle to the explanation. We follow the now well-known step of taking an image as input:

3.5.2 Problems with Fast RCNN

But even Fast RCNN has certain problem areas. It also uses selective search as a proposal method to find the Regions of Interest, which is a slow and time-consuming process. It takes around 2 seconds per image to detect objects, which is much better compared to RCNN. But when we consider large real-life datasets, then even a Fast RCNN doesn't look so fast anymore.

But there's yet another object detection algorithm that trump Fast RCNN. And something tells me you won't be surprised by its name

3.6 Understanding Faster RCNN

3.6.1 Intuition of Faster RCNN

Faster RCNN is the modified version of Fast RCNN. The major difference between them is that Fast RCNN uses selective search for generating Regions of Interest, while Faster RCNN uses "Region Proposal Network", aka RPN. RPN takes image feature maps as an input and generates a set of object proposals, each with an objectness score as output. The below steps are typically followed in a Faster RCNN approach: We

take an image as input and pass it to the ConvNet which returns the feature map for that image. Region proposal network is applied on these feature maps. This returns the object proposals along with their objectness score. A RoI pooling layer is applied on these proposals to bring down all the proposals to the same size. Finally, the proposals are passed to a fully connected layer which has a SoftMax layer and a linear regression layer at its top, to classify and output the bounding boxes for objects. To begin with, Faster RCNN takes the feature maps from CNN and passes them on to the Region Proposal Network. RPN uses a sliding window over these feature maps, and at each window, it generates k Anchor boxes of different shapes and sizes. Anchor boxes are fixed sized boundary boxes that are placed throughout the image and have different shapes and sizes. For each anchor, RPN predicts two things:

The first is the probability that an anchor is an object (it does not consider which class the object belongs to),Second is the bounding box regressor for adjusting the anchors to better fit the object. We now have bounding boxes of different shapes and sizes which are passed on to the RoI pooling layer. Now it might be possible that after the RPN step, there are proposals with no classes assigned to them. We can take each proposal and crop it so that each proposal contains an object. This is what the RoI pooling layer does. It extracts fixed sized feature maps for each anchor, then these feature maps are passed to a fully connected layer which has a SoftMax and a linear regression layer. It finally classifies the object and predicts the bounding boxes for the identified objects.

3.6.2 Problems with Faster RCNN

All the object detection algorithms we have discussed so far use regions to identify the objects. The network does not look at the complete image in one go but focuses on parts of the image sequentially. This creates two complications:

The algorithm requires many passes through a single image to extract all the objects, as there are different systems working one after the other, the performance of the systems further ahead depends on how the previous systems performed

3.6.3 Summary of the Algorithms covered

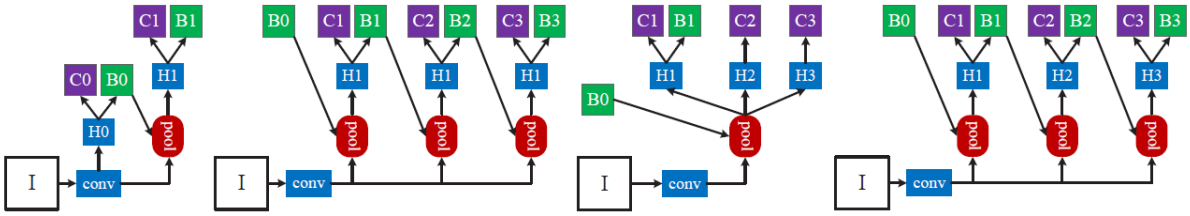
The below table is a nice summary of all the algorithms we have covered in this chapter.

Algorithm	Features	Prediction time / image	Limitations
CNN	Divides the image into multiple regions and then classify each region into various classes.	N/A	Needs a lot of regions to predict accurately and hence high computation time.
RCNN	Uses selective search to generate regions. Extracts around 2000 regions from each image.	40-50 Seconds	High computation time as each region is passed to the CNN separately also it uses three different model for making predictions.
Fast RCNN	Each image is passed only once to the CNN and feature maps are extracted. Selective search is used on these maps to generate predictions. Combines all the three models used in RCNN together.	2 Seconds	Selective search is slow and hence computation time is still high.
Faster RCNN	Replaces the selective search method with region proposal	0.2 Seconds	Object proposal takes time and as there are different systems

	network which made the algorithm much faster		working one after the other, the performance of systems depends on previous.

3.7 Cascade R-CNN

we extend the two-stage architecture of the Faster R-CNN ,shown in Figure below. The first stage is a proposal sub-network (“H0”), applied to the entire image, to produce preliminary detection hypotheses, known as object proposals. In the second stage, these hypotheses are then processed by a region-of-interest detection sub-network (“H1”), denoted as detection head. A final classification score (“C”) and a bounding box (“B”) are assigned to each hypothesis. We focus on modeling a multistage detection sub-network, and adopt, but are not limited to, the RPN [30] for proposal detection



(a) Faster R-CNN (b) Iterative BBox at inference (c) Integral Loss (d) Cascade R-CNN

Figure 3. The architectures of different frameworks. “I” is input image, “conv” backbone convolutions, “pool” region-wise feature extraction. “H” network head. “B” bounding box. and “C” classification. “B0” is proposals in all architectures.

Figure 56:cascaded RCNN

the Cascade R-CNN, for the design of high-quality object detectors. This architecture was shown to avoid the problems of overfitting at training and quality mismatch at inference. The solid and consistent detection improvements of the Cascade R-CNN on the challenging COCO and the popular PASCAL VOC datasets suggest the modeling and understanding of various concurring factors are required to advance object detection. The Cascade RCNN was shown to be applicable to many object detection architectures. We believe that it can be useful to many future object detection research efforts

P.O.C	YOLO	SSD	R-CNN	FAST R-CNN	FASTER R-CNN
Accuracy	Low	Low	Very Low	Intermediate	High
Speed	High	Intermediate	Very Low	Low	High
Implementation	Easy	Complex	Easy	Intermediate	Intermediate

3.8 Full Faster R-CNN implementation

To train the architecture we used we need to put our dataset in a file format called tfrecord which is facilitated by a website called Roboflow, Roboflow makes managing, preprocessing, augmenting, and versioning datasets for computer vision seamless. Developers reduce 50% of their code when using Roboflow's workflow, automate annotation quality assurance, save training time, and increase model reproducibility.

This website is used to change from one format to another to the annotation files and this will be used to generate the formats we will need in training in case you need to train on your custom dataset, We documented the steps that are needed to change the format and to train the architecture In our GitHub repo

https://github.com/Ramy-osama/Hardware-FasterRCNN

This is the implementation for a hardware Faster-RCNN neural network architecture used in object detection in autonomous cars we will divide this project into 2 parts The first part is the software implementation of our Faster-RCNN neural network The second part of the code is the hardware implementation of the backbone network of the Faster-RCNN (VGG-16) to fasten the simulation The hardware implementation will be only used in inference mode and we will use the software implementation to get the correct weights we will use in the hardware part

Software implementation

Introduction

- We will be simulating the Faster-RCNN architecture using google colabs as this will fasten the training using google servers

- we will use a unique custom dataset which is a subset of the VOCtrainval_11-May-2012 that contains objects that is present on the road normally
- This data set is found in the dataset folder where it is divided into 2 folders one which contains the images and the annotation files(XML format) and the other folder contains the tfrecord format that will be used in training (explained later)

About Roboflow for Data Management

Roboflow makes managing, preprocessing, augmenting, and versioning datasets for computer vision seamless. Developers reduce 50% of their code when using Roboflow's workflow, automate annotation quality assurance, save training time, and increase model reproducibility.

This website is used to change from one format to another to the annotation files and this will be used to generate the formats we will need in training in case you need to train on your custom dataset (note the dataset attached in the dataset folder already has the correct format) for the full tutorial on how to work with roboflow with your custom dataset please check the following link Roboflow

Folder structure

The software folder contains 2 items and 2 folders:

1. Models contains the actual Faster-RCNN model being implemented
2. tensorflow-object-detection-faster-rcnn contains some .ipynb files that we can use to train the dataset on google colab
3. Faster-RCNN.ipynb this is the modified file that we will use for inference or training the Faster-RCNN model

4. frozen_inference_graph.pb this is the file that will be outputed from the Faster-RCNN.ipynb after training the model using the dataset in the folder, This file will be used in inference mode

Inference mode

1. Open Faster-RCNN.ipynb
2. Run this code snippet

```
[ ]  !pip install tensorflow_gpu==1.15
```

3. Run this code snippet

- Number of evalutation steps isn't important when running the inference mode
- If you run this code snippet and doing the training mode i suggest doing a small number of steps as a start ie. num_steps = 100 for example
- The extracted frozen_graph.pb attached was done on 1000 steps because the number of the attached images isn't big enough so we don't run on the problem of overfitting

- Make sure that the selected_model = 'faster_rcnn_inception_v2' to choose the Faster-RCNN model

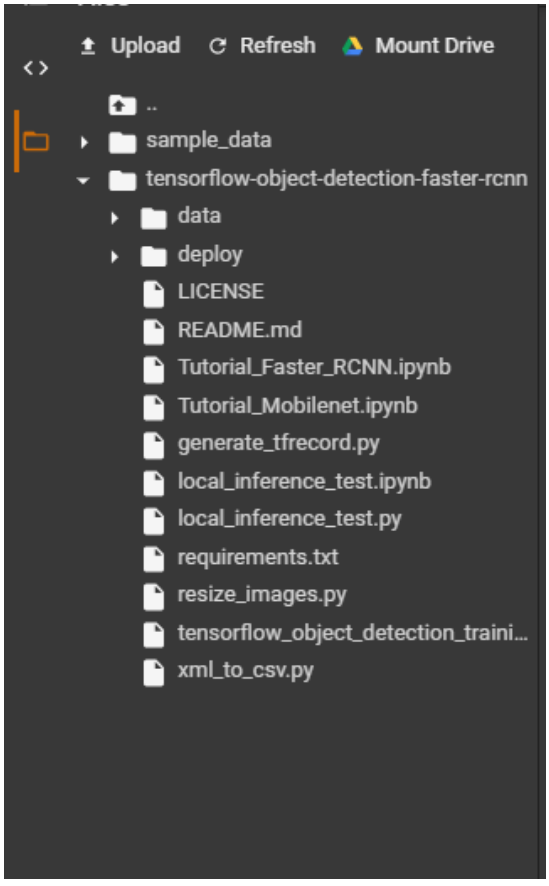
```
[ ]
    # Number of training steps - 1000 will train very quickly, but more steps will increase accuracy.
    num_steps = 10000  # 200000 to improve

    # Number of evaluation steps.
    num_eval_steps = 50

    MODELS_CONFIG = {
        'ssd_mobilenet_v2': {
            'model_name': 'ssd_mobilenet_v2_coco_2018_03_29',
            'pipeline_file': 'ssd_mobilenet_v2_coco.config',
            'batch_size': 12
        },
        'faster_rcnn_inception_v2': {
            'model_name': 'faster_rcnn_inception_v2_coco_2018_01_28',
            'pipeline_file': 'faster_rcnn_inception_v2_pets.config',
            'batch_size': 12
        },
        'rfcn_resnet101': {
            'model_name': 'rfcn_resnet101_coco_2018_01_28',
            'pipeline_file': 'rfcn_resnet101_pets.config',
            'batch_size': 8
        }
    }

    # Pick the model you want to use
    # Select a model in `MODELS_CONFIG`.
    selected_model = 'faster_rcnn_inception_v2'

    # Name of the object detection model to use.
    MODEL = MODELS_CONFIG[selected_model]['model_name']

    # Name of the pipline file in tensorflow object detection API.
    pipeline_file = MODELS_CONFIG[selected_model]['pipeline_file']

    # Training batch size fits in Colabe's Tesla K80 GPU memory for selected model.
    batch_size = MODELS_CONFIG[selected_model]['batch_size']
```

4. Run this code snippet

```
import os

%cd /content

repo_dir_path = os.path.abspath(os.path.join('.', os.path.basename(repo_url)))

!git clone {repo_url}
%cd {repo_dir_path}
!git pull
```

```
/content
Cloning into 'tensorflow-object-detection-faster-rcnn'...
remote: Enumerating objects: 885, done.
remote: Total 885 (delta 0), reused 0 (delta 0), pack-reused 885
Receiving objects: 100% (885/885), 24.84 MiB | 38.30 MiB/s, done.
Resolving deltas: 100% (418/418), done.
/content/tensorflow-object-detection-faster-rcnn
Already up to date.
```

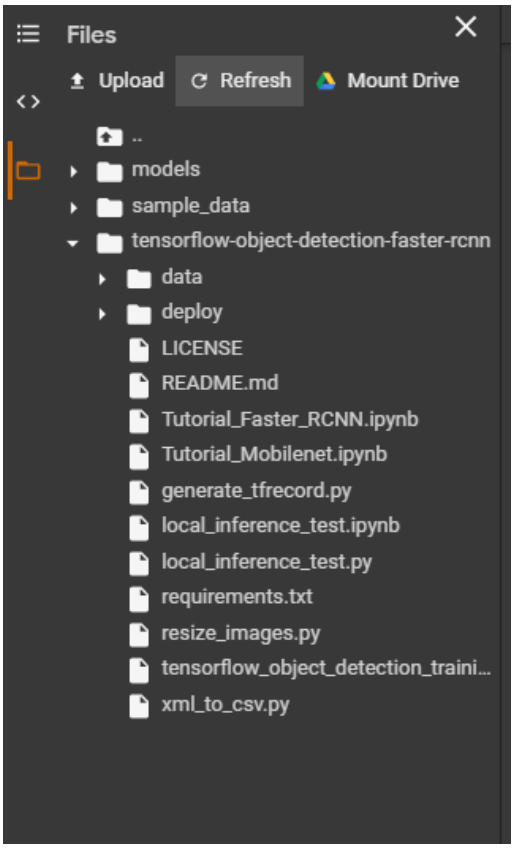
5. This should be your folder after the last step



6. Run this code snippet

```
%cd /content
!git clone --quiet https://github.com/tensorflow/models.git

!apt-get install -qq protobuf-compiler python-pil python-lxml python-tk

!pip install -q Cython contextlib2 pillow lxml matplotlib

!pip install -q pycocotools

%cd /content/models/research
!protoc object_detection/protos/*.proto --python_out=.

import os
os.environ['PYTHONPATH'] += ':/content/models/research/:/content/models/research/slim/'

!python object_detection/builders/model_builder_test.py
```

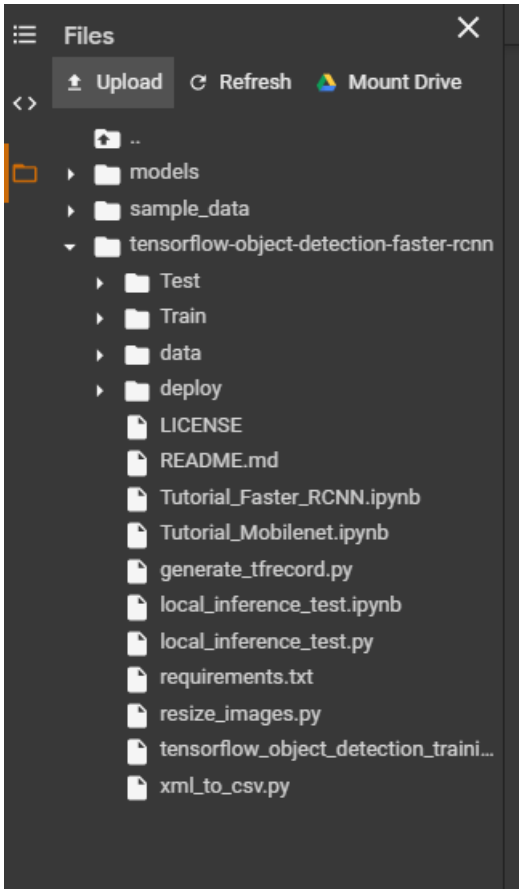
7. This should be your folder after the last step (Model folder should be added)



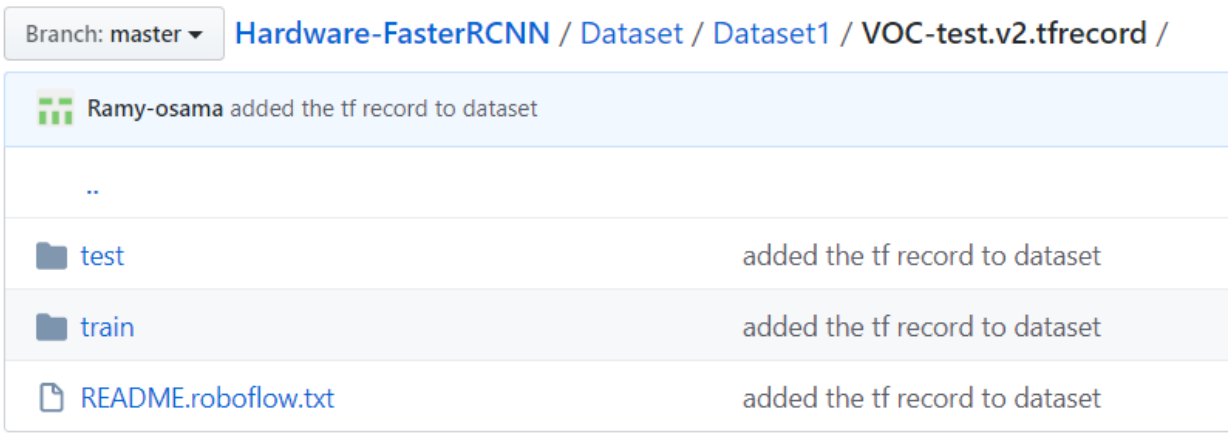
8. Run this code snippet (may remove it)

```
[8]  %cd /content/tensorflow-object-detection-faster-rcnn/data
     /content/tensorflow-object-detection-faster-rcnn/data
```

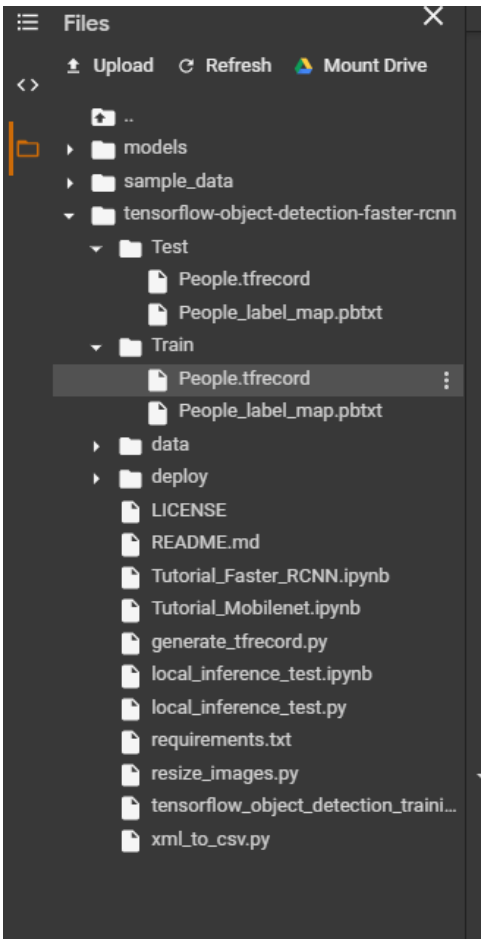
9. Create 2 directories named Test and Train as shown in the following screenshot



10. check the Dataset\Dataset1\VOC-test.v2.tfrecord folder where you can see the test and train directories



11. Add the content in the directories to the directories created in step 9 after this step it should look something like this



12. we will work with the following code snippet in this step

```
[ ]  # NOTE: Update these TFRecord names from "cells" and "cells_label_map" to your files!
     test_record_fname = '/content/tensorflow-object-detection-faster-rcnn/data/valid/cells.tfrecord'
     train_record_fname = '/content/tensorflow-object-detection-faster-rcnn/data/train/cells.tfrecord'
     label_map_pbtxt_fname = '/content/tensorflow-object-detection-faster-rcnn/test/cells_label_map.pbtxt'
```

- In the first variable we should copy the path of the tfrecord file in the test directory we created in step 9
- In the second variable we should copy the path of the tfrecord file in the train directory we created in step 9
- In the third variable we should copy the path of the pbtxt file in the test directory we created in step 9
- Note: in order to copy the path simply right click on the file you want and then paste in the code snippet
- After the following steps this should be the shape of the code snippet

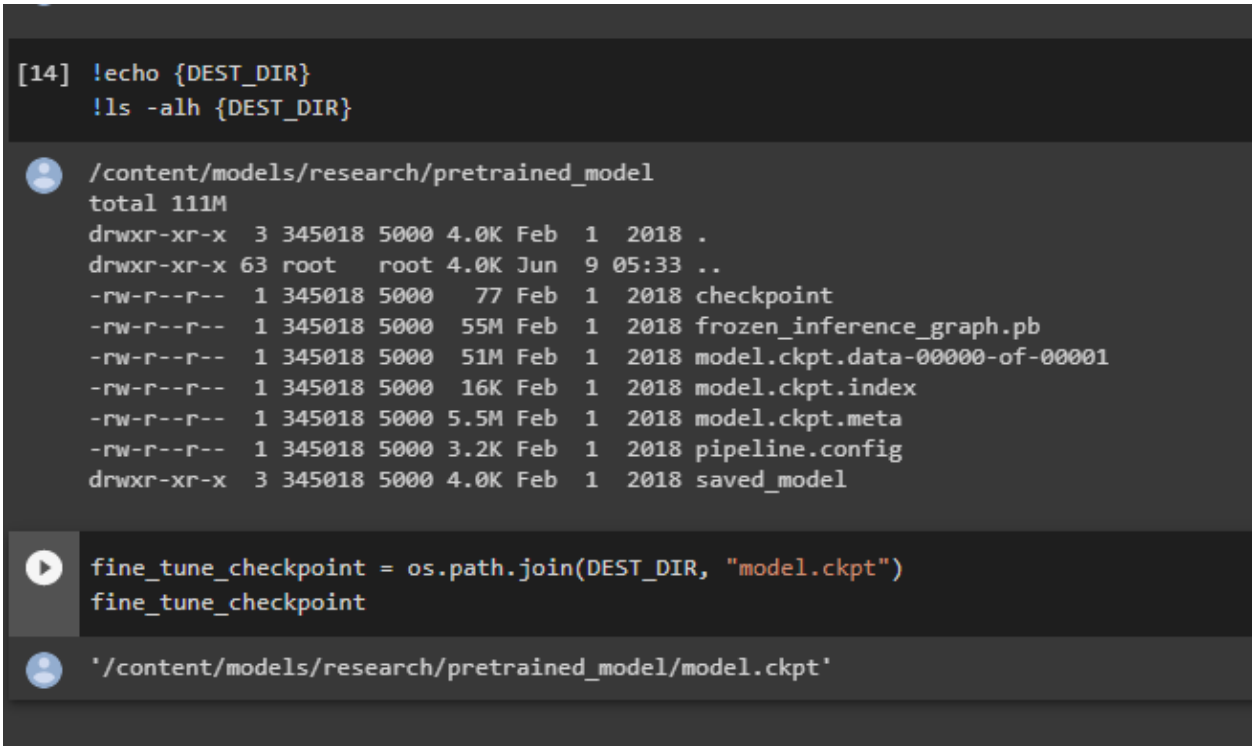
```
     # NOTE: Update these TFRecord names from "cells" and "cells_label_map" to your files!
     test_record_fname = '/content/tensorflow-object-detection-faster-rcnn/Test/People.tfrecord'
     train_record_fname = '/content/tensorflow-object-detection-faster-rcnn/Train/People.tfrecord'
     label_map_pbtxt_fname = '/content/tensorflow-object-detection-faster-rcnn/Test/People_label_map.pbtxt'
```

13. Run the code snippet

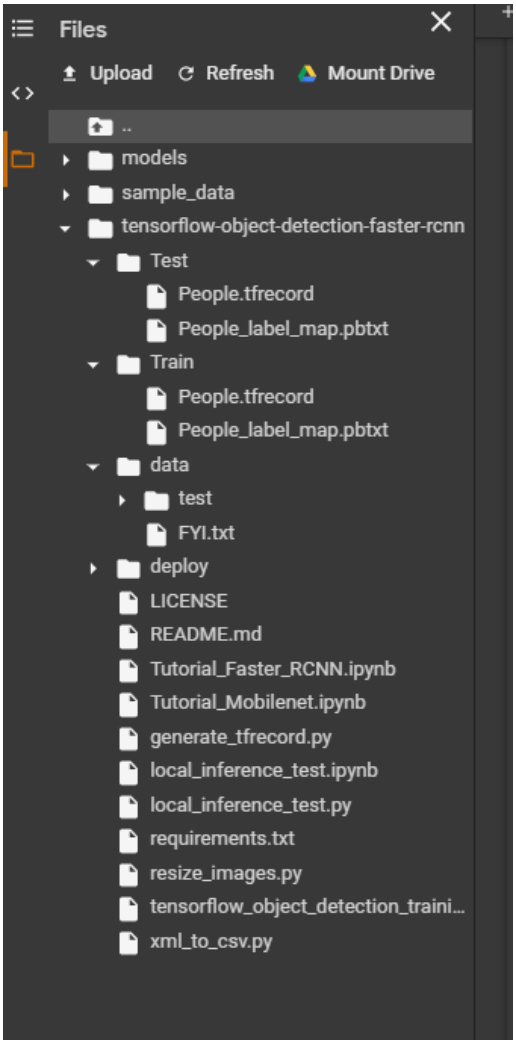
14. Run this code snippet

```
[ ]  %cd /content/models/research

     import os
     import shutil
     import glob
     import urllib.request
     import tarfile
     MODEL_FILE = MODEL + '.tar.gz'
     DOWNLOAD_BASE = 'http://download.tensorflow.org/models/object_detection/'
     DEST_DIR = '/content/models/research/pretrained_model'

     if not (os.path.exists(MODEL_FILE)):
         urllib.request.urlretrieve(DOWNLOAD_BASE + MODEL_FILE, MODEL_FILE)

     tar = tarfile.open(MODEL_FILE)
     tar.extractall()
     tar.close()

     os.remove(MODEL_FILE)
     if (os.path.exists(DEST_DIR)):
         shutil.rmtree(DEST_DIR)
     os.rename(MODEL, DEST_DIR)
```

15. Run these code snippets

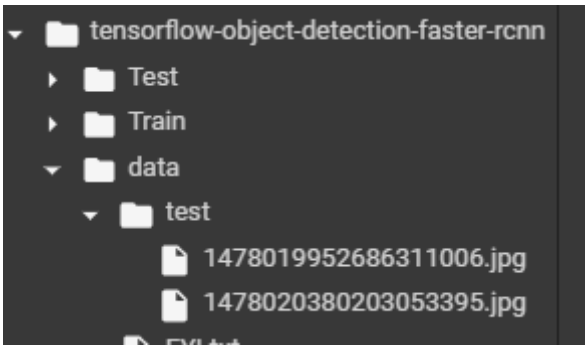


16. Create a new sub directory to the data directory named test after this step the folders should look like this

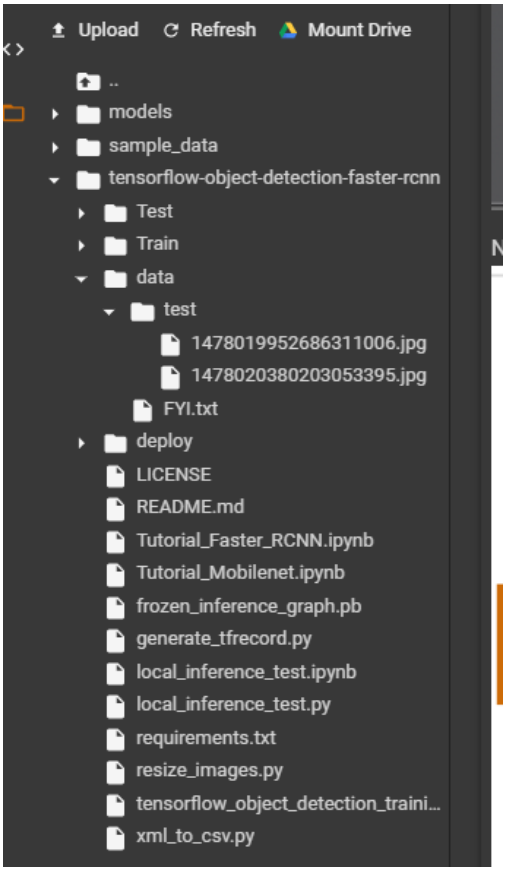


17. Go to Dataset/Dataset2/subest_udacity_dataset where you can see the images we will use for testing these images are a subset of the udacity dataset

18. upload the images you want to test in the inference mode to the test directory created in step 16 from the directory stated in step 17 your folders may look like this after uploading



19. Upload the frozen_inference_graph.pb file found in software directory to the tensorflow-object-detection-faster-rcnn folder in google colabs your folders should look something like this(note the added pb file)



20. Now check this code snippet

```
[ ]  import os
     import glob

     # Path to frozen detection graph. This is the actual model that is used for the object detection.
     PATH_TO_CKPT ='/content/models/research/pretrained_model/frozen_inference_graph.pb'

     # List of the strings that is used to add correct label for each box.
     PATH_TO_LABELS = label_map_pbtxt_fname

     # If you want to test the code with your images, just add images files to the PATH_TO_TEST_IMAGES_DIR.
     PATH_TO_TEST_IMAGES_DIR =  os.path.join(repo_dir_path, "data/test")

     assert os.path.isfile(PATH_TO_CKPT)
     assert os.path.isfile(PATH_TO_LABELS)
     TEST_IMAGE_PATHS = glob.glob(os.path.join(PATH_TO_TEST_IMAGES_DIR, "*.*"))
     assert len(TEST_IMAGE_PATHS) > 0, 'No image found in `{}`.'.format(PATH_TO_TEST_IMAGES_DIR)
     print(TEST_IMAGE_PATHS)
```

- Delete the frozen_inference_graph.pb in the following directory /content/models/research/pretrained_model/frozen_inference_graph.pb

- Upload the new frozen_inference_graph.pb found in the software directory in github

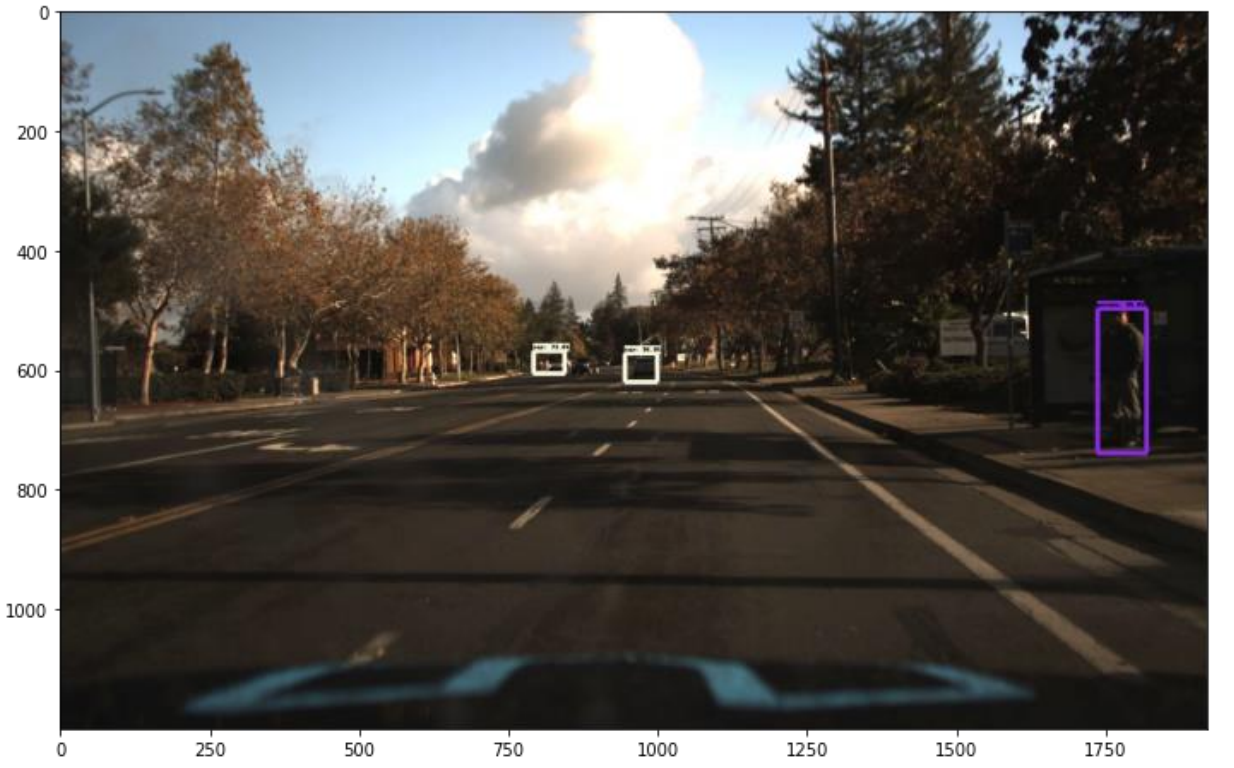
21. Run the code snippet

22. add the following 2 lines at the start of this code snippet then run it !pip install tf-slim
    num_classes = get_num_classes(label_map_pbtxt_fname)

```
[ ]  %cd /content/models/research/object_detection

     import numpy as np
     import os
     import six.moves.urllib as urllib
     import sys
     import tarfile
     import tensorflow as tf
     import zipfile

     from collections import defaultdict
     from io import StringIO
     from matplotlib import pyplot as plt
     from PIL import Image

     # This is needed since the notebook is stored in the object_detection folder.
     sys.path.append("..")
     from object_detection.utils import ops as utils_ops


     # This is needed to display the images.
     %matplotlib inline


     from object_detection.utils import label_map_util

     from object_detection.utils import visualization_utils as vis_util
```

23.Run the remaining code snippets and enjoy the results!

Results


0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750


0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750


0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750


0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750
STOP

3.9 VGG-16

In this part we build a full VGG-16 neural network using keras we used took the code in [11] as a reference for our architecture, at first we trained our neural network using the fashion Mnist dataset to make sure that our methodology in extracting weights to the hardware part is correct, the following table shows the architecture used.

Table 1 VGG-16 architecture

Layer (Type)	Output Shape	Param #
Convd2d	(None,28,28,64)	640
Convd2d_1	(None,28,28,64)	36928
Max_pooling2d	(None,14,14,64)	0
Convd2d_2	(None,14,14,128)	73856
Convd2d_3	(None,14,14,128)	147584
Max_pooling2d_1	(None,7,7,128)	0
Convd2d_4	(None,7,7,256)	295168
Convd2d_5	(None,7,7,256)	590080
Convd2d_6	(None,7,7,256)	590080
Max_pooling2d_2	(None,4,4,256)	0
Convd2d_7	(None,4,4,512)	1180160
Convd2d_8	(None,4,4,512)	2359808
Convd2d_9	(None,4,4,512)	2359808
Max_pooling2d_3	(None,2,2,512)	0
Convd2d_10	(None,2,2,512)	2359808
Convd2d_11	(None,2,2,512)	2359808
Convd2d_12	(None,2,2,512)	2359808
Max_pooling2d_4	(None,1,1,512)	0
Flatten	(None,512)	0
Dense	(None,4096)	2101248
Dense_1	(None,4096)	16781312
Dense_2	(None,10)	40970

But we faced a problem when training a full VGG-16 on the fashion mnist dataset as the images in this dataset are of size 28x28 so when passed to the many extraction layers provided in the VGG-16 architecture (convolution, Relu, maxpooling) the size of the feature map will be very small so we can see from the results of training in Table 3 the neural network didn’t train.

Table 2 Results of training the fashion mnist dataset on VGG-16 architecture

NO.	EPOCH	LOSS	ACCURACY
1	1/10	2.3029	0.0989
2	2/10	2.3028	0.0988
3	3/10	2.3028	0.0971
4	4/10	2.3028	0.0965
5	5/10	2.3028	0.0988
6	6/10	2.3028	0.0990
7	7/10	2.3028	0.0979
8	8/10	2.3028	0.0990
9	9/10	2.3028	0.0992
10	10/10	2.3028	0.0988

3.9.1 Light weight VGG-16 architecture

Due to hardware memory constraints to download weights on it and due to RAM constraints, that is needed to extract the weights from software, A light weight VGG-16 was designed that has much less number of parameters while also having the high accuracy to classifying the fashion mnist dataset that is used to verify that the process of extraction of weights and the blocks constructed in hardware is working properly, the architecture is shown in Table 4

Table 3 lightweight VGG-16 architecture

Layer (type)	Output Shape	Param #
Conv2d	(None, 28, 28, 64)	640
Conv2d_1	(None, 28, 28, 64)	36928
Conv2d_2	(None, 28, 28, 64)	36928
Conv2d_3	(None, 28, 28, 64)	36928
Max_pooling2d	(None, 14, 14, 64)	0
Conv2d_4	(None, 14, 14, 64)	36928
Conv2d_5	(None, 14, 14, 64)	36928
Max_pooling2d_1	(None, 7, 7, 64)	0
flatten	(None, 3136)	0
dense	(None, 1024)	3212288
dense_1	(None, 1024)	1049600
dense_2	(None, 10)	10250

The result of training for 20 epochs was nearly 99% and it achieved 92.4% accuracy after testing it against the test dataset

3.9.2 Extraction of weights

After training the CNN, weights are extracted from the layers into a text file that is read in our C++ codes to build our architecture in hardware, we use some Python scripts to make the output format readable and compatible with the fixed point calculations we will use in hardware, also some more scripts was developed to change the images to an array form so It can be easily stored in memory in the hardware part

3.10 Hardware implementation

The hardware implementation main purpose in this work and in the field of the deep neural networks is the acceleration, the acceleration of CNNs -that requires high complexity and computations to achieve the required accuracy- can be solved by using GPUs, but this solution comes not only with a great cost but also a high power consumption, therefore we are targeting FPGA for implementing the accelerating as it can compromise between the power consumption and the speed of the CNN that is necessary for real time applications [13].

Designing the CNN on FPGA introduces many challenges, one of them is the design method, there is a tradeoff between the main two design methods which are RTL and HLS, the RTL direct HW design may achieve higher efficiency but it consumes time and poses complexity for large designs, on the other hand the HLS method offers quick development and provides many automated features like pipelining.

In this work we use the HLS based design approach as it fits our requirement for fast implementation, reasonable efficiency and accuracy within the available resources.

We use Vivado HLS tool and as shown in figure 4 the flow of the design is to write codes in high level language as C or C++ with considering some limitation and instructions and the tool generates VHDL and Verilog codes that can be synthesized, the tools offers the ability of simulating wave forms, and we can target different FPGAs like ZYNQ directly and specify the type of memory easily (FIFIO, ROM, BRAM, RAM).

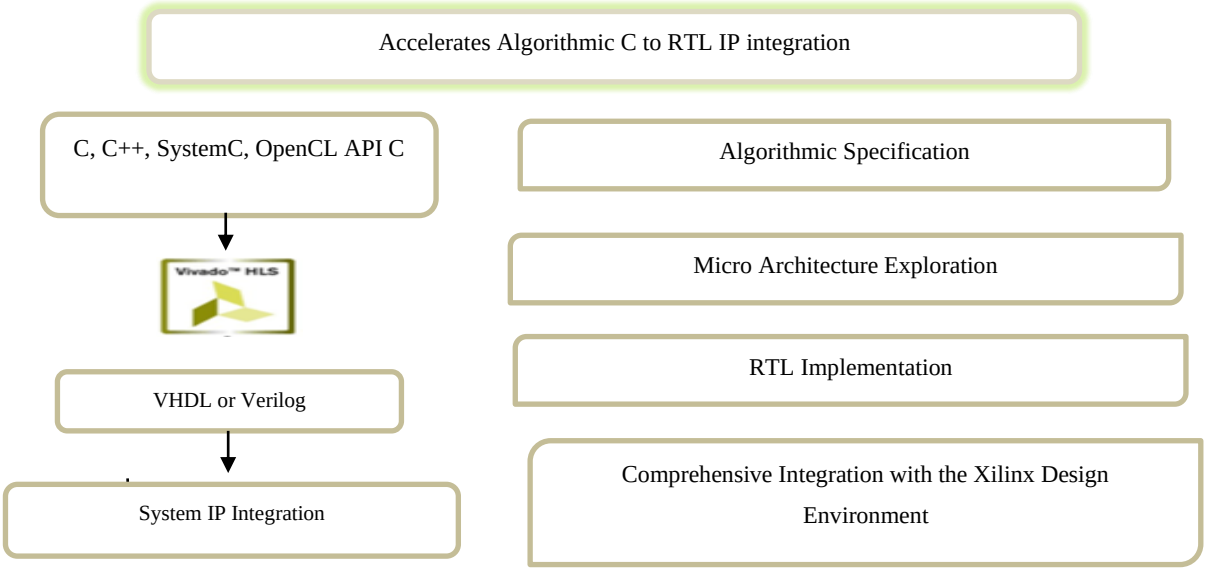


Figure 4: Flow of the HLS design approach using (Vivado HLS).

One other challenge for the CNN design on FPGA is the limited recourses, as known, computations in neural networks are performed traditionally with floating point calculations using either GPU or CPU but when it comes to hardware implementation, using floating point calculation doesn't just make it slower due to the difficulty of controlling the mantissa and the exponent for various operations [12] but also consumes the resources, and to avoid this problem we use fixed point calculations that use much less resources and doesn't affect the performance [1].

Table 4: FPGA resource consumption comparison for multiplier and adder with different types of data.

	Xilinx Logic			
	Multiplier		Adder	
	LUT	FF	LUT	FF
Fp32	708	858	430	749
Fp16	221	303	211	337
Fixed32	1112	1143	32	32
Fixed16	289	301	16	16
Fixed8	75	80	8	8

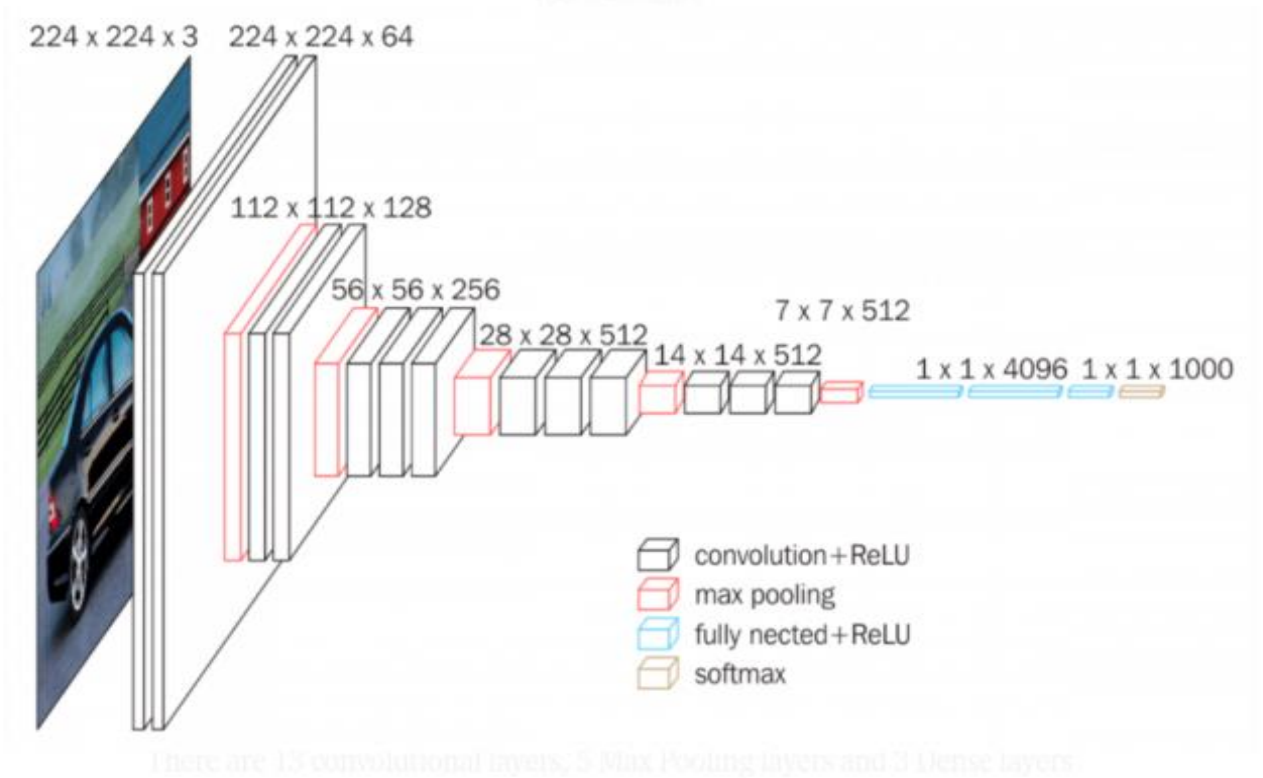


Figure 5 Block diagram of layers of VGG-16.

- As shown in figure 5 the architecture is divided into the following layers:

3.10.1 Convolution Layer.

Inputs:

Inputs of this layer are as follows:

Three-dimensional array, two of them represents the size of the image or the feature map which depends on in which layer we are, and the third dimension is the number of channels, for example, the first layer always takes a 3 channels image (RGB).

Four-dimensional array which represents number of kernels, number of channels, and two dimensions for the size.

One-dimensional array of the bias of every kernel applied.

Outputs:

The output is a three-dimensional array of the feature maps which is passed to Relu and Max-pooling layers afterwards. Figure 5 shows the simulation result of convolution layer.

3.10.2 Padding.

Inputs:

This is implemented before the convolution, so the input is either the image or the feature map and the main aim of this it is to preserve the image size after convolution so that we stick to the vgg16 CNN architecture.

Outputs:

The output is the padded image and then it will be passed to the convolution layer.

3.10.3 Max Pooling Layer.

Inputs:

Retrieve the output of the convolution layer after the Relu operation then halve each dimension by 2 which reduce the number of values to one quarter of the original number (by using a 2x2 window with stride of 2 to avoid any overlap)

Outputs:

The output is the feature maps that contain the maximum value of each four-pixel values in the original input, their number will be equal to the number of the filters and they will be passed to the next convolution layer.

3.10.4 RELU Layer.

Inputs:

This layer applies the ReLU operation which is nonlinear operation used to reduce the number of deep layers in the CNN and it basically operates on each value and remove all negative values from the output feature map before Maxpooling layer.

3.10.5 Fully Connected Layer.

Inputs:

This layer applies the Flattening operation in which it takes the output of the previous layer (convolution layer) which is a 3 dimensional array,

Outputs:

The fully-connected layer “flattens” the previous layer and turns it into single vector which is 1 dimensional array that can be an input for the next stage

3.10.6 Integration of (Padding-Convolution –Max pooling).

This layer is integration between the previous layers which work as first layer of vgg16 where all operations of Padding convolution and Maxpooling are applied after each other in this order

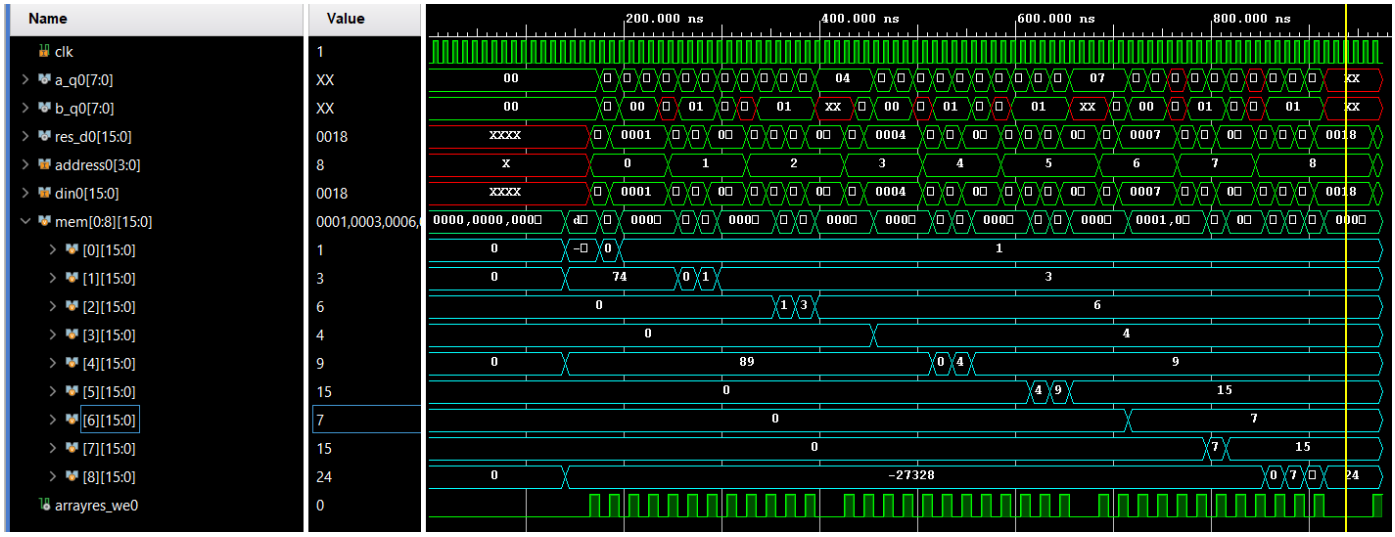
- Start the function by padding the input image so the feature map after convolution is same size of input image
- do the convolution with one image RGB , two filters with three channels each
- Do max pooling 2x2 which reduce the output feature map size by 2.

A. An illustrative example

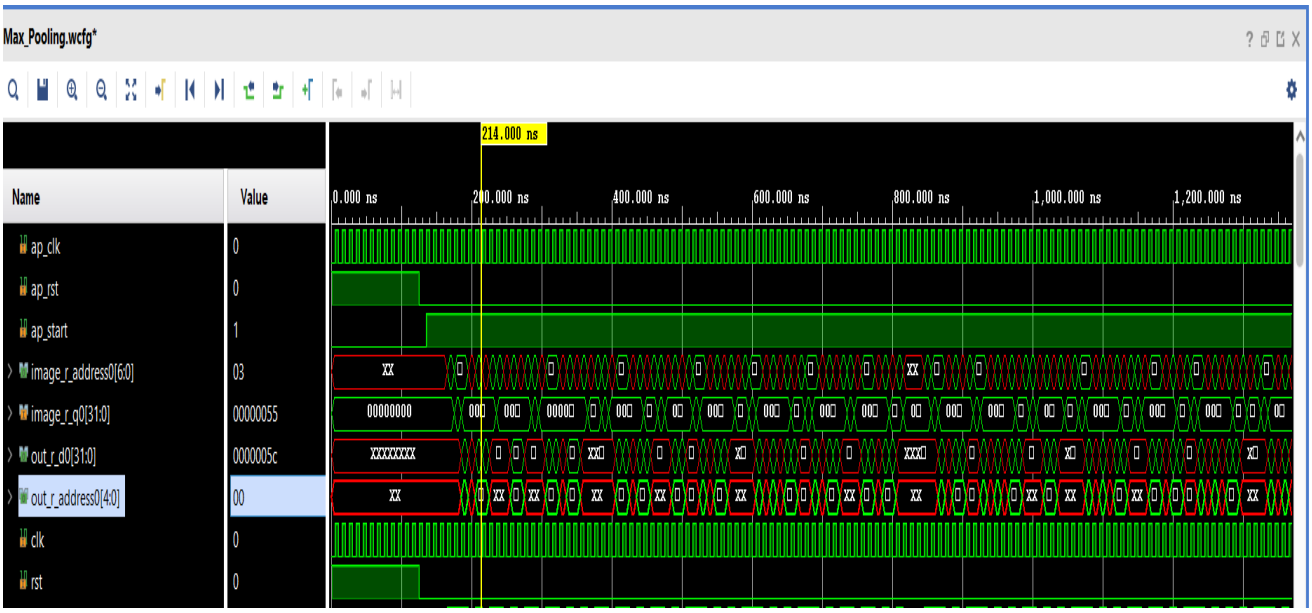
Matrix multiplication is the first example we experimented using the Vivado tool as it is one of the main operations in the project.

Figure 6 Matrix multiplication Simulation Result.

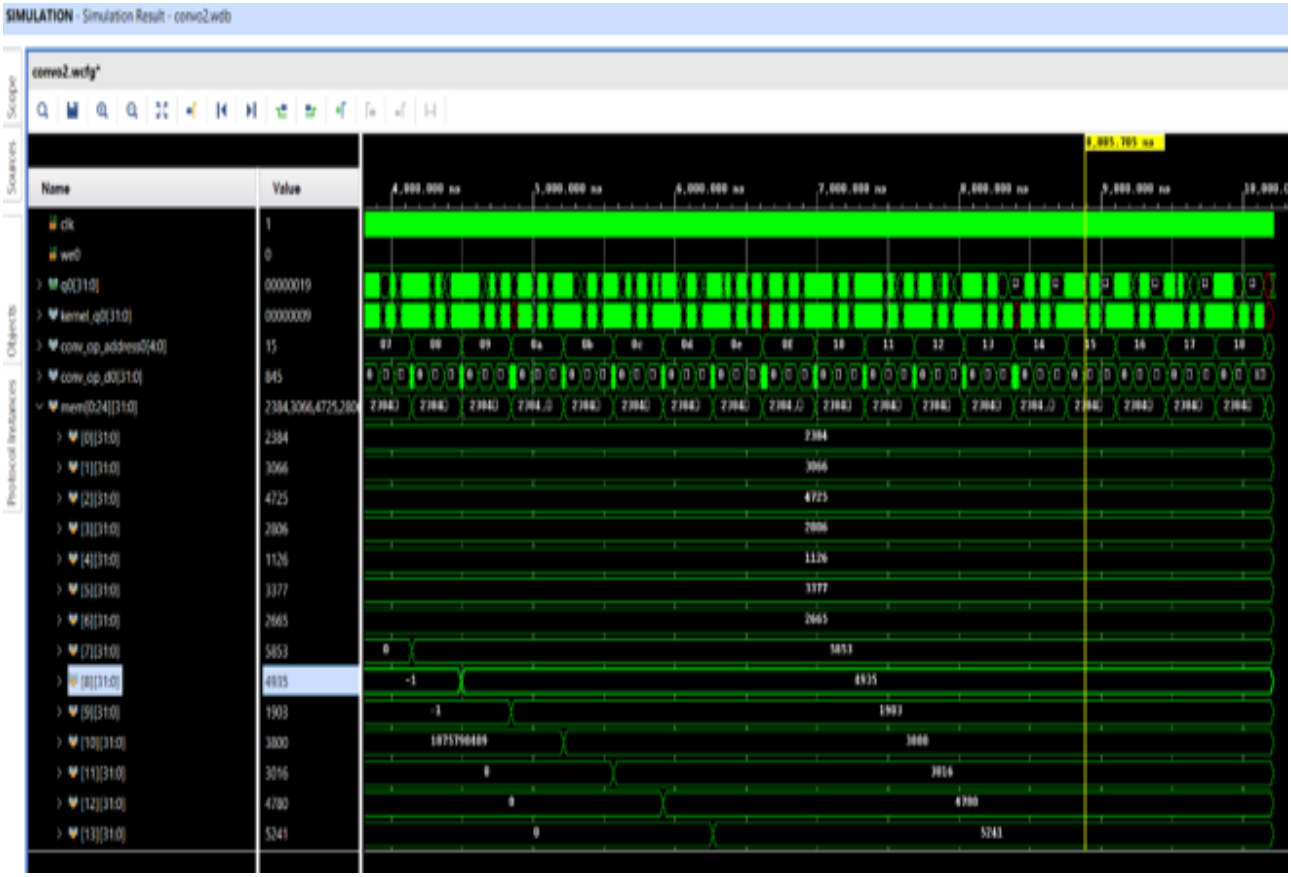
In Figure. 6, the memory signals in the aqua color is output array that has the res_d0 saved in it. It shows clearly that it is the expected output. By tracing the wave form of inputs and the result the timing of everything goes as expected. This operation is basic one in the convolution layer and it gave us a good insight on how to use the tool power like pipelining, optimization techniques and memory implementation. In addition to that the report generated from tool about the resources needed on the FPGA to carry out this process also by referring to table 2 it shows clearly the estimated latency of multiplying 3x3 matrix and generating the output matrix and save it in a memory.

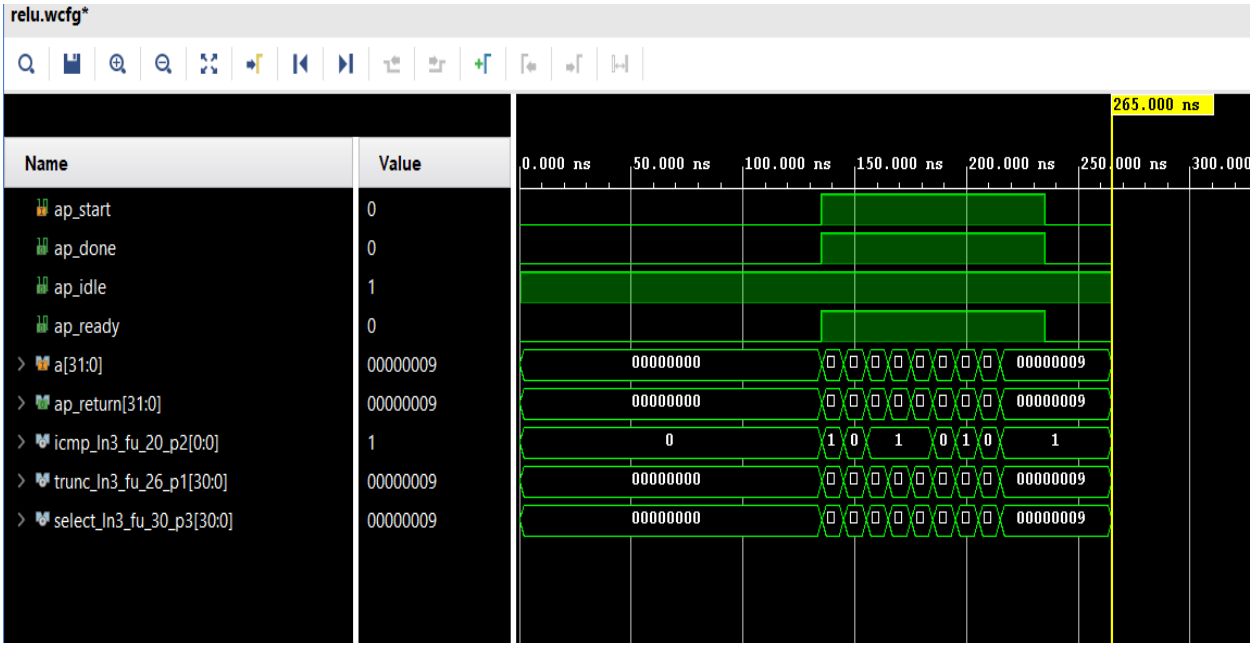


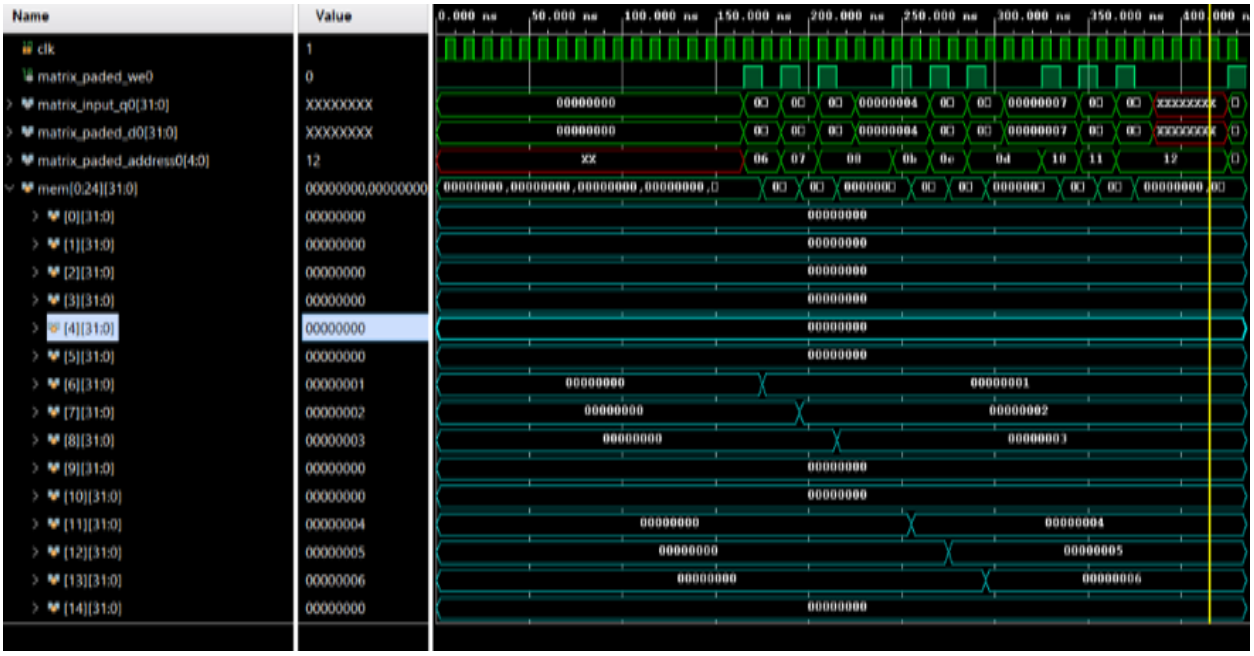
3.11. Hardware Results

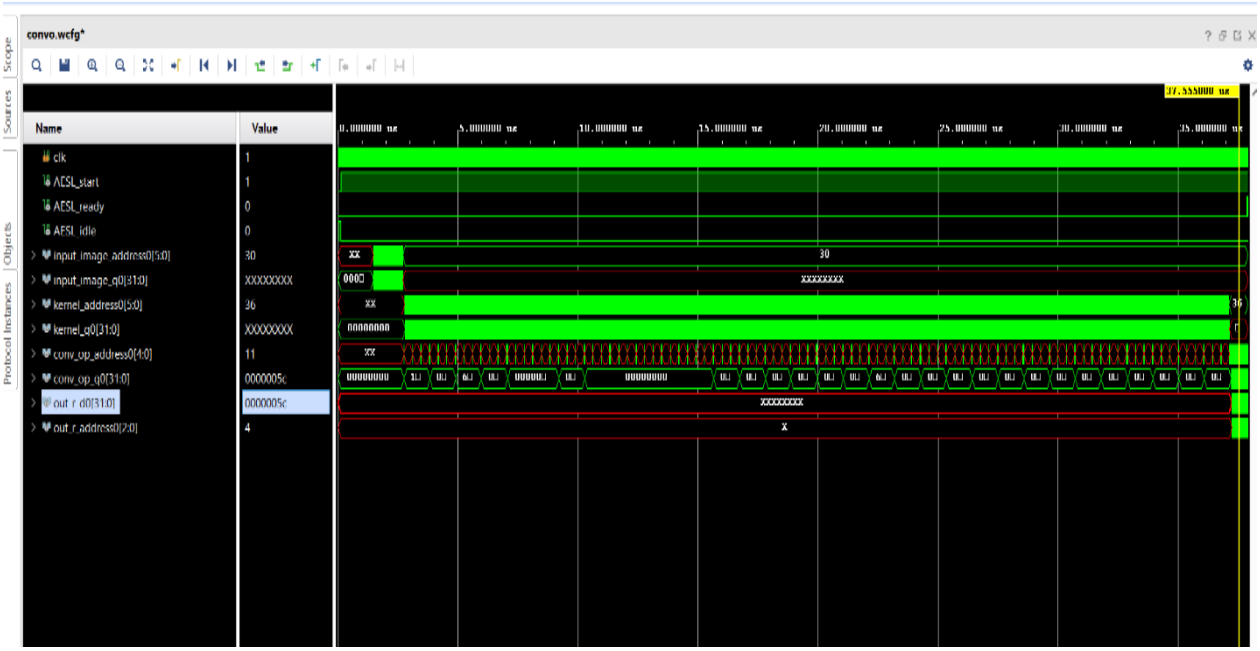

Max_Pooling.wcfg*
214.000 ns
Name
Value
0.000 ns
200.000 ns
400.000 ns
600.000 ns
800.000 ns
1,000.000 ns
1,200.000 ns
ap_clk
0
ap_rst
0
ap_start
1
image_r_address0[6:0]
03
image_r_q0[31:0]
00000055
out_r_d0[31:0]
0000005c
out_r_address0[4:0]
00
clk
0
rst
0

.


SIMULATION - Simulation Result - convo2.wdb
convo2.wcfg*
Name
Value
clk
1
we0
0
q0[31:0]
00000019
kernel_q0[31:0]
00000009
conv_op_address0[4:0]
15
conv_op_d0[31:0]
845
mem[0:24][31:0]
2384,3066,4725,280
[0][31:0]
2384
[1][31:0]
3066
[2][31:0]
4725
[3][31:0]
2806
[4][31:0]
1126
[5][31:0]
3377
[6][31:0]
2665
[7][31:0]
5853
[8][31:0]
4935
[9][31:0]
1903
[10][31:0]
3800
[11][31:0]
3016
[12][31:0]
4700
[13][31:0]
5241


relu.wcfg*
Name
Value
ap_start
ap_done
ap_idle
ap_ready
a[31:0]
ap_return[31:0]
icmp_ln3_fu_20_p2[0:0]
trunc_ln3_fu_26_p1[30:0]
select_ln3_fu_30_p3[30:0]
0
0
1
0
00000009
00000009
1
00000009
00000009
0.000 ns
50.000 ns
100.000 ns
150.000 ns
200.000 ns
265.000 ns
300.000


Name
Value
clk
matrix_paded_we0
matrix_input_q0[31:0]
matrix_paded_d0[31:0]
matrix_paded_address0[4:0]
mem[0:24][31:0]
[0][31:0]
[1][31:0]
[2][31:0]
[3][31:0]
[4][31:0]
[5][31:0]
[6][31:0]
[7][31:0]
[8][31:0]
[9][31:0]
[10][31:0]
[11][31:0]
[12][31:0]
[13][31:0]
[14][31:0]
1
0
XXXXXXXX
XXXXXXXX
12
00000000,00000000
00000000
00000001
00000002
00000003
00000004
00000005
00000006



3.11.1 Padding

By referring to the Figure 8 it shows that the function output is just as expected and after the required clock cycles the correct values are on the output. Furthermore, the output is being saved in the memory once they are ready on the output and before the memory address changes to the next one the true value should be ready to write in the memory. Tracing this waveform enabled us to ensure that the function worked correct.

3.11.2 Convolution

After the padding function comes the convolution which do the most calculations and need the largest time and resources. Figure 9 shows the simulation of this function on hardware. The write signal is only high when it is correct value is being set on the output.

3.11.3 Max pooling

The Max pooling function differs from the previous functions as it doesn't go through the feature maps row by row, but it uses a window 2x2 which is dealing with two different rows at a time. So, in the wave form the output may go high impedance at some points but the output array is still acting as intended (Figure 10).

3.11.4 RELU

The RELU simulation was the easiest as it only converting the negative values to zero and leave the other values as it is.

3.11.5 Fully Connected

The simulation shows that the 3d array of the output layer of convolution is flattened with the same values to a 1d array which can be passed to other layers

3.11.6 Integrated layer

After simulation of all these layers we needed to integrate them in one function to ensure that all of them are working smoothly with each other especially the interface between them .the simulation in figure 12 shows it takes a lot of time before the output is being ready and calculated that explains the high impedance value at the output lasting this long

3.12 Performance evaluation

3.12.1 Performance Metrics

Table 5 Latency of each Layer.

The proposed architecture	Latency(cycles)		Latency(absolute)	
	min	max	min	min
	869	869	8.690 us	8.690 us
Padding	79	79	0.790 us	0.790 us
Max pooling	206	206	206	206
RELU	0	0	0	0
Fully-Connected				
Integration of padding & convolution & max pooling	3575	3789	37.570 us	37.890 us

The metrics of performance in our design are the speed mainly of performing the convolution calculations as it gets much more time than other layers. Table 4 indicates the latency of different layers it shows clearly that convolution is the most time consuming. Noting that cascading layers together and increasing the latency time that must be taken in consideration as we are not using the real image size yet nor the exact number of filters in VGG16.

3.12.2 The utilization in FPGA

The utilization of the design when simulating it to know how much resources it will use from the available on the FPGA up till now the design is not complete, so we are using very small resources (table 5).

Table 6 Comparison between Utilization of each Layer.

	Total					Utilization (%)				
	BRAM_18K	DSP48E	FF	LUT	URAM	BRAM_18K	DSP48E	FF	LUT	URAM
Convolution	0	3	476	757	0	0	0	0	0	0
Padding	0	0	59	277	0	0	0	0	0	0
Max pooling	0	0	471	1541	0	0	0	0	0	0
RELU	0	0	0	49	0	0	0	0	0	0
Fully Connected										
Integration of padding, convolution and max pooling	1	3	944	2186	0	0	0	0	0	0

3.13 Comparison with related work

Table 7 Hardware accelerators

Hardware accelerator	Execution time (ms)(forward path)
Proposed architecture	40.94
Pascal titan x	5.32
GTX 1080	7

Table 9 Clock cycles

HW Function	Accelerated Clock cycles
Conv layer 3	$2.147\ X10^9$
FC layer 6	$2.147\ X10^9$
FC layer 7	$1.649X10^9$
FC layer 8	$0.4034\ X10^9$

Table 10 Total LUT, DSP and FF

Conv layer1	LUT's	BRAMs 36k	DSPs	FFs
Optimized RTL	32,968	288	105	9210
SDSOC	19,260	477	95	11,037

3.13.1 Our work

Table 11 Clock cycles

Clock	Target	Estimated	Uncertainty
Ap_clk			

Table 12 Total DSP, FF and LUT

Name	BRAM 18K	DSP48E	FF	LUT	URAM
DSP					
Expression					
FIFO					
Instance					
Memory					
Multiplexer					
Register					
Total					
Available					
Utilization (%)					

3.13.2 Simple comparison.

The previous tables summarize the total clock cycles, total flip-flops, look up tables and digital signal processors of our work-related work in [17].

3.14 Limitations

Due to implementation on hardware we are always limited with memory so we can't do larger architectures which has larger number of parameter which may have high accuracy, also we may need large number of hardware to implement any architecture which may not be available in commercial FPGAs

Chapter four: Summary and future work

4.1 Summary

Let's summarize what we did in this semester:

- We discovered the different types of neural networks and we chose the convolutional neural networks because it is of best use in object detection as it can extract features from the images we use as a dataset
- We learnt about all the layers included in a convolutional neural network and the mathematics behind it as we will need it in the future work when we start to create our chip
- We got familiar with the tools we need when we start to do our simulations
- We have seen the suggested architectures in the convolutional neural networks, and we chose the faster RCNN as it is the best according to our constrains
- We started to build our code for the chosen architecture

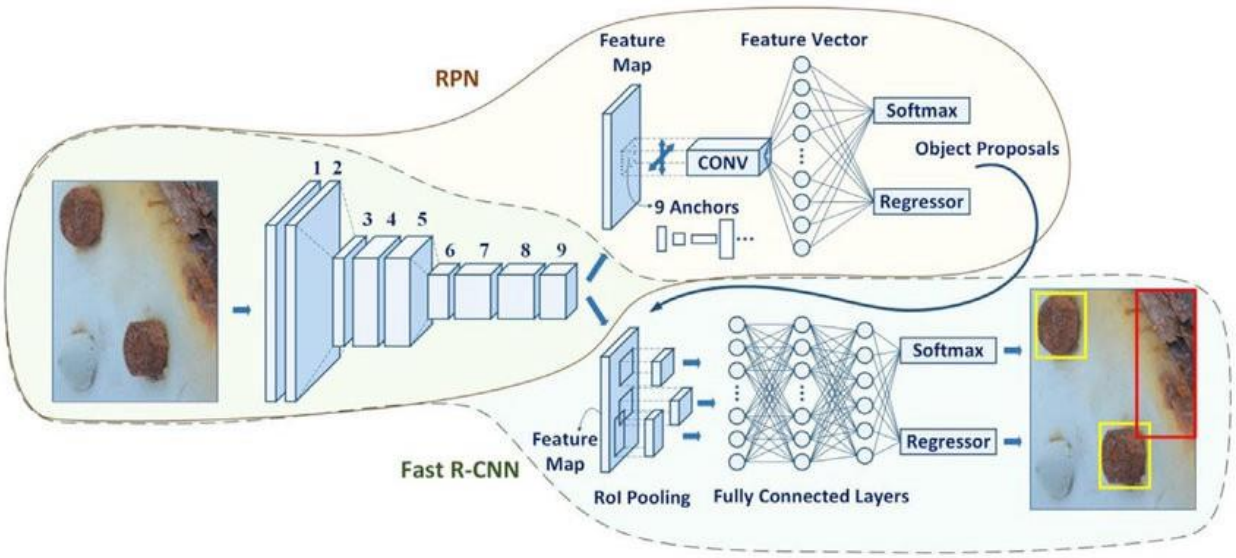


Figure 57

4.2 Future Work

- Run Co-Simulation between the hardware and software part
- Download the code on FPGA
- Construct the whole Faster-RCNN in hardware

References

[1] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. 2017. [DL] A Survey of FPGA-Based Neural Network Inference Accelerator

[2] J. Duarte et al., “Fast inference of deep neural networks in fpgas for particle physics,” arXiv preprint arXiv: 1804.06913, 2018.

[3] J. Qiu et al., “Going deeper with embedded fpga platform for convolutional neural network,” in Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA ’16. New York, NY, USA: ACM, 2016, pp. 26–35. [Online].

[4] Joseph Fiowa ,TensorFlow-object-detection-faster-rcnn,(2013),GitHub

[5] C. L. Zitnick and P. Dollar, “Edge boxes: Locating object ´ proposals from edges,” in European Conference on Computer Vision (ECCV), 2014.

[6] Shaoqing Ren, PY-faster-rcnn) 2015), GitHub: https://github.com/rbgirshick/py-faster-rcnn

[7] K. He, X. Zhang, S. Ren, and J. Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. In ECCV, 2014

[8] D. Baptista, F. Morgado-Dias and L. Sousa, "A Platform based on HLS to Implement a Generic CNN on an FPGA," 2019 International Conference in Engineering Applications (ICEA), Sao Miguel, Portugal, 2019, pp. 1-7, doi: 10.1109/CEAP.2019.8883473.

[9] S. Ghaffari and S. Sharifian, "FPGA-based convolutional neural network accelerator design using high level synthesize," 2016 2nd International Conference of Signal Processing and Intelligent Systems (ICSPIS), Tehran, 2016, pp. 1-6, doi: 10.1109/ICSPIS.2016.7869873.

[10] K. Simonyan and A. Zisserman, “Very Deep

Convolutional Networks for Large-Scale Image

[11] Rohit, VGG16-In-Keras (2018),GitHub: https://github.com//1297rohit/VGG16-In-Keras

[12] Yongmei Zhou and Jingfei Jiang, "An FPGA-based accelerator implementation for deep convolutional neural networks," 2015 4th International Conference on Computer Science and Network Technology (ICCSNT), Harbin, 2015,pp.829832,doi:10.1109/ICCSNT.2015.7490869.

[13] Hassan, R.O., Mostafa, H. Implementation of deep neural networks on FPGA-CPU platform using Xilinx SDSOC. *Analog Integr Circ Sig Process* (2020).

[14] L. D. LeCun, Y.; Boser, B.; Denker, J. S.;

Henderson, D.; Howard, R. E.; Hubbard, W.; Jackel,"Backpropagation Applied to Handwritten Zip CodeRecognition," Neural Computation, vol. 1, no. 4. Pp.541–551, 1989.

[15] Y. LeCun and L. Bottou, "Sn: A simulator for connectionist models," in Proceedings of NeuroNimes 88, 1988.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," Adv. Neural Inf. Process. Syst., pp. 1–9, 2012.

[17] Rania O. Hassan1 • Hassan Mostafa1, 2

"Implementation of deep neural networks on FPGA-CPU platform using Xilinx SDSOC"