Bachelor of Science in Computer Science and Engineering

Image Classification using Deep Convolutional Neural Networks

by

Sabbir Ahmed (134443)

Md. Saadman Farhad (134434)

Department of Computer Science and Engineering (CSE)
Islamic University of Technology (IUT)
a subsidiary organ of the Organization of Islamic Cooperation (OIC)
Gazipur-1704, Dhaka, Bangladesh

ISLAMIC UNIVERSITY OF TECHNOLOGY

Image Classification using Deep Convolutional Neural Networks

Author:
Sabbir Ahmed (134443)

Md. Saadman Farhad (134434)

Supervisor:
Md. Hasanul Kabir, Ph.D.
Associate Professor
Department of Computer Science and Engineering (CSE)
Islamic University of Technology (IUT)

A thesis submitted to the Department of Computer Science and Engineering (CSE) in partial fulfilment of the requirements for the degree of B.Sc. in CSE

Department of Computer Science and Engineering (CSE)
Islamic University of Technology (IUT)
a subsidiary organ of the Organization of Islamic Cooperation (OIC)
Gazipur-1704, Dhaka, Bangladesh

November 2017

Candidates' Declaration

It is hereby declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma.

Signature of the candidate	Signature of the candidate
Md. Saadman Farhad	Sabbir Ahmed
Student no. 134434	Student no. 134443
Department: CSE	Department: CSE

Signature of the Supervisor

Md. Hasanul Kabir, Ph.D.

Associate Professor

Abstract

Image classification is the task of taking an input image and outputting a class (a cat, dog, etc.) or a probability of classes that best describes the image. For humans, this task of recognition is one of the first skills we learn from the moment we are born and is one that comes naturally and effortlessly as adults. Without even thinking twice, we’re able to quickly and seamlessly identify the environment we are in as well as the objects that surround us. When we see an image or just when we look at the world around us, most of the time we are able to immediately characterize the scene and give each object a label, all without even consciously noticing. These skills of being able to quickly recognize patterns, generalize from prior knowledge, and adapt to different image environments are ones that we do not share with our fellow machines. Convolutional neural networks, Sounds like a weird combination of biology and math with a little CS sprinkled in, but these networks have been some of the most influential innovations in the field of computer vision. 2012 was the first year that neural nets grew to prominence as Alex Krizhevsky used them to win that year’s ImageNet competition (basically, the annual Olympics of computer vision), dropping the classification error record from 26% to 15%, an astounding improvement at the time. Ever since then, a host of companies have been using deep learning at the core of their services.

In our research we experimentd on image classification using different deep learning frameworks. The following sections describes basics of deep learning and how it can be used in case of image classification and convolutional neural networks. We have also discussed the different deep learning frameworks and current applications. Finally we shared our gained results and knowledge. This will help the researchers to get a clear idea about getting knowledge in the field of image classification with deep convolutional neural networks.

Acknowledgements

It is an auspicious moment for me to submit our thesis work by which we are eventually going to end our Bachelor of Science study. At the beginning, we want to express our heart-felt gratitude to Almighty Allah for his blessings to bestow upon us which made it possible to complete this thesis research successfully. Without the mercy of Allah, we wouldn’t be where we are right now. All thanks and praises be to Allah.

Secondly, we would like to thank our thesis supervisor, Dr. Md. Hasanul Kabir, Associate Professor, CSE, IUT, for his support and guidance on this thesis. Dr. Kabir has been an instrumental to this work and our careers. He taught us how to do research, think critically, be a graduate student, and teach effectively. His all-time guidance, encouragement and continuous observation made the whole matter as a successful one. Without his continuous support, this thesis would not see the path of proper itinerary of the research world.

It was our pleasure to get the cooperation and coordination Md. Redwan Karim Sony, Lecturer, Department of CSE, IUT for his enormous help in directing us towards our goal and helping us in time of our need. We are grateful to him for his constant and energetic guidance, constructive criticism and valuable advice.

The faculty members of the CSE department of IUT helped make our working environment a pleasant one, by providing a helpful set of eyes and ears when problems arose.

We would like to thank the jury members of my thesis committee for the many interesting comments and criticism that helped improve this manuscript. Lastly, we are deeply grateful to our friends and family for their unconditional support. This work would have never been completed without the consistent support and encouragement from them throughout the program.

Contents

List Of Figures

Chapter 1

1 Introduction

In this chapter, we first present an overview of our thesis that includes the significance of the problem and the problem statement in detail. Besides, we also discuss about the different research challenges what we are going to face in the whole scenario.

1.1 Overview

When a computer sees an image (takes an image as input), it will see an array of pixel values. Depending on the resolution and size of the image, it will see a 32 x 32 x 3 array of numbers (The 3 refers to RGB values). Just to drive home the point, let's say we have a color image in JPG form and its size is 480 x 480. The representative array will be 480 x 480 x 3. Each of these numbers is given a value from 0 to 255 which describes the pixel intensity at that point. These numbers, while meaningless to us when we perform image classification, are the only inputs available to the computer. The idea is that we give the computer this array of numbers and it will output numbers that describe the probability of the image being a certain class (.80 for cat, .15 for dog, .05 for bird, etc.).

Now that the problem as well as the inputs and outputs are known, let's think about how to approach this. What we want the computer to do is to be able to differentiate between all the images it's given and figure out the unique features that make a dog a dog or that make a cat a cat. This is the process that goes on in our minds subconsciously as well. When we look at a picture of a dog, we can classify it as such if the picture has identifiable features such as paws or 4 legs. In a similar way, the computer is able perform image classification by looking for low level features such as edges and curves, and then building up to more abstract concepts through a series of convolutional layers. This is a general overview of what a CNN does.

1.2 Problem Statement:

Deep learning in image classification problem is a relatively new field. A lot of ongoing research is taking place in improving the use of deep learning to classify images with further accuracy [2]. Giant tech companies are using huge amount of data present at their disposal to train complex models. So basically our task is the same as classifying images based on set of categories after the network has been trained on a dataset.

1.3 Research Challenges

Deep learning is now a very attractive field and vastly used by the giant tech companies around the world for various applications. Although deep learning ensures huge promise in the field of image processing, it comes with some basic problems which requires attention. To get the best out of deep learning a large amount of data is required, if only thousands of data is available then deep learning is highly unlikely to outperform other approaches. [8]

Deep learning is computationally expensive to train. Even with high end GPUs, complex models take weeks to train using hundreds of machines.

Coming to a particular decision in deep learning is quite confusing. Determining the topology/flavor/training method/ hyper parameters for deep learning is a black art with no proper theory for guidance.

1.4 Applications of Deep Learning

There is a lot of excitement around artificial intelligence, machine learning and deep learning at the moment. It is also an amazing opportunity to get on the ground floor of some really powerful technology. The following discussion includes some of the attractive applications of deep learning:

1.4.1 Automatic Colorization of Black and White Images

Image colorization is the problem of adding color to black and white photographs. Traditionally this was done by hand with human effort because it is such a difficult task.

Deep learning can be used to use the objects and their context within the photograph to color the image, much like a human operator might approach the problem. [1]

A visual and highly impressive feat. This capability leverages of the high quality and very large convolutional neural networks trained for ImageNet and co-opted for the problem of image colorization. Generally the approach involves the use of very large convolutional neural networks and supervised layers that recreate the image with the addition of color.

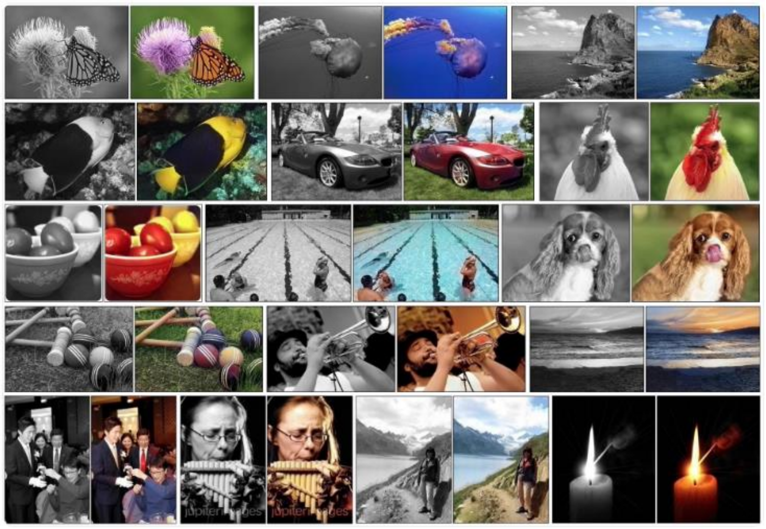


Figure 1.1: Automatic Colorization of Black and White Images [14]

1.4.2 Automatically Adding Sounds to Silent Movies

In this task the system must synthesize sounds to match a silent video.

The system is trained using 1000 examples of video with sound of a drum stick striking different surfaces and creating different sounds. A deep learning model associates the video frames with a database of pre-rerecorded sounds in order to select a sound to play that best matches what is happening in the scene. [2]

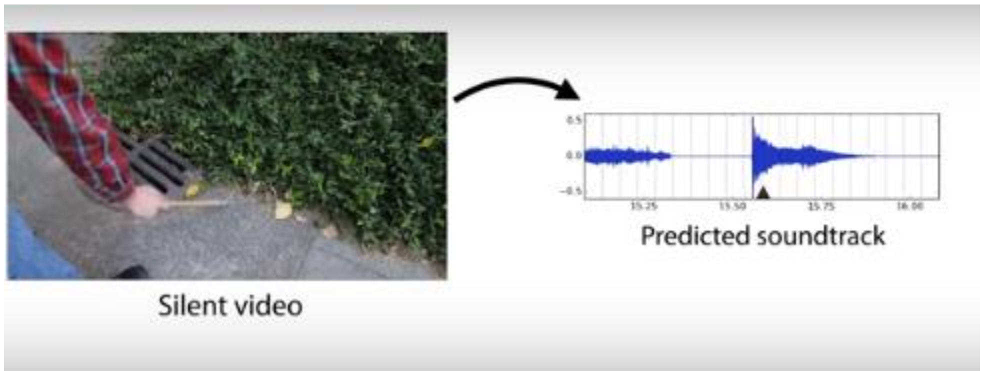


Figure 1.2: Automatically Adding Sounds to Silent Movies [14]

1.4.3 Automatic Machine Translation

This is a task where given words, phrase or sentence in one language, automatically translate it into another language. [3]

Automatic machine translation has been around for a long time, but deep learning is achieving top results in two specific areas: Automatic Translation of Text, Automatic Translation of Images. [7]

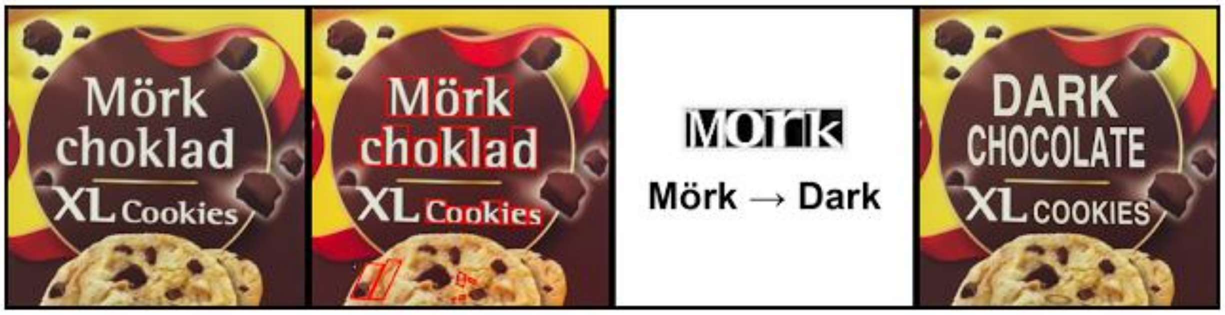


Figure 1.3: Automatic Machine Translation [14]

1.4.4 Object Classification and Detection in Photographs

This task requires the classification of objects within a photograph as one of a set of previously known objects.

State-of-the-art results have been achieved on benchmark examples of this problem using very large convolutional neural networks. [4]

A breakthrough in this problem by Alex Krizhevsky et al. results on the ImageNet classification problem called AlexNet.

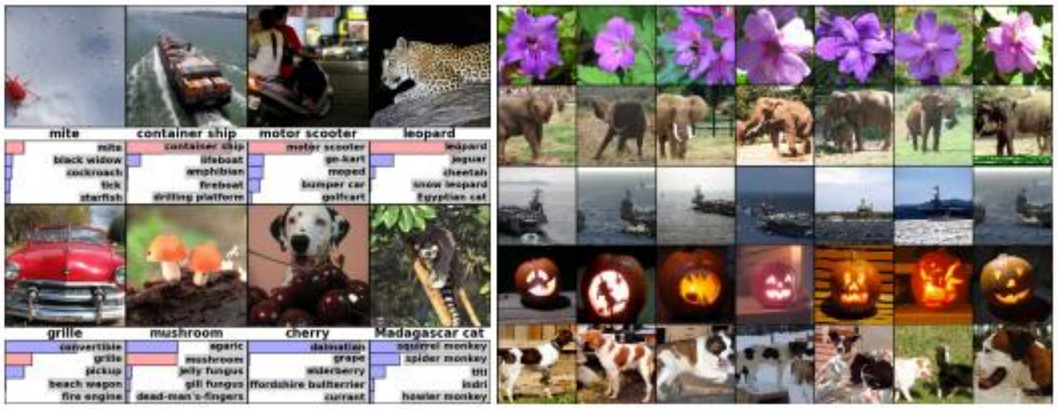


Figure 1.3: Object Classification and Detection in Photographs [14]

A more complex variation of this task called object detection involves specifically identifying one or more objects within the scene of the photograph and drawing a box around them.

1.4.5 Automatic Handwriting Generation

This is a task where given a corpus of handwriting examples, generate new handwriting for a given word or phrase. [5]

The handwriting is provided as a sequence of coordinates used by a pen when the handwriting samples were created. From this corpus the relationship between the pen movement and the letters is learned and new examples can be generated ad hoc.

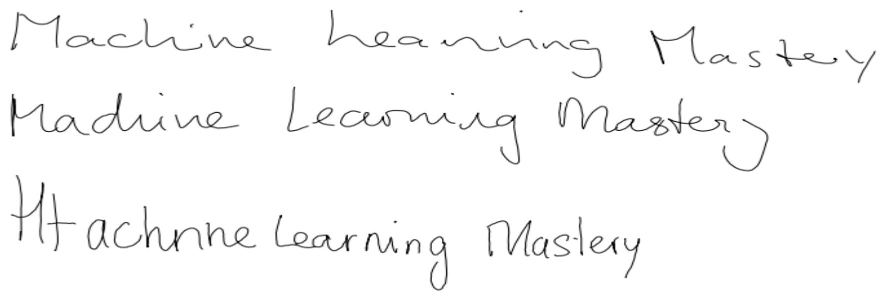


Figure 1.4: Automatic Handwriting Generation [14]

1.4.6 Automatic Image Caption Generation

Automatic image captioning is the task where given an image the system must generate a caption that describes the contents of the image. In 2014, there were an explosion of deep learning algorithms achieving very impressive results on this problem, leveraging the work from top models for object classification and object detection in photographs. [6]

Once the object is deleted from the photograph, the next step can be assigning a coherent sentence description to the image. Generally, the systems involve the use of very large convolutional neural networks for the object detection in the photographs and then a recurrent neural network like an LSTM to turn the labels into a coherent sentence.

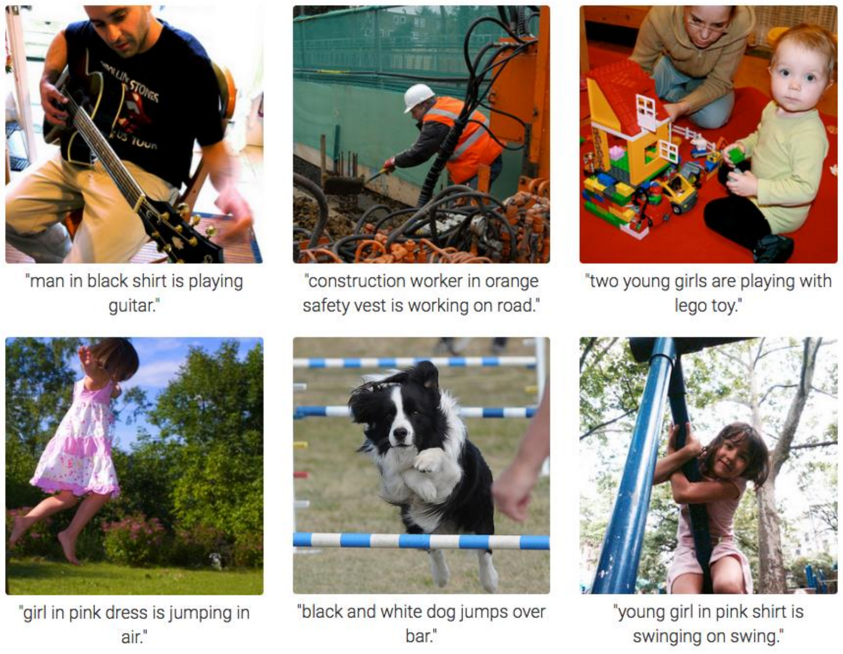

"man in black shirt is playing guitar."
"construction worker in orange safety vest is working on road."
"two young girls are playing with lego toy."
"girl in pink dress is jumping in air."
"black and white dog jumps over bar."
"young girl in pink shirt is swinging on swing."

Figure 1.6: Automatic Image Caption Generation [14]

1.5 Challenges in Deep Learning:

Deep Learning has become one of the primary research areas in developing intelligent machines. Most of the well-known applications (such as Speech Recognition, Image Processing and NLP) of AI are driven by Deep Learning. Deep Learning algorithms mimic human brains using artificial neural networks and progressively learn to accurately solve a given problem. But there are significant challenges in Deep Learning systems which we have to look out for.

1.5.1 Lots and Lots of data:

Deep learning algorithms are trained to learn progressively using data. Large data sets are needed to make sure that the machine delivers desired results. As human brain needs a lot of experiences to learn and deduce information, the analogous artificial neural network requires copious amount of data. The more powerful abstraction, the more parameters need to be tuned and more parameters require more data.

1.5.2 Overfitting in Neural Networks

At times, the there is a sharp difference in error occurred in training data set and the error encountered in a new unseen data set. It occurs in complex models, such as having too many parameters relative to the number of observations. The efficacy of a model is judged by its ability to perform well on an unseen data set and not by its performance on the training data fed to it

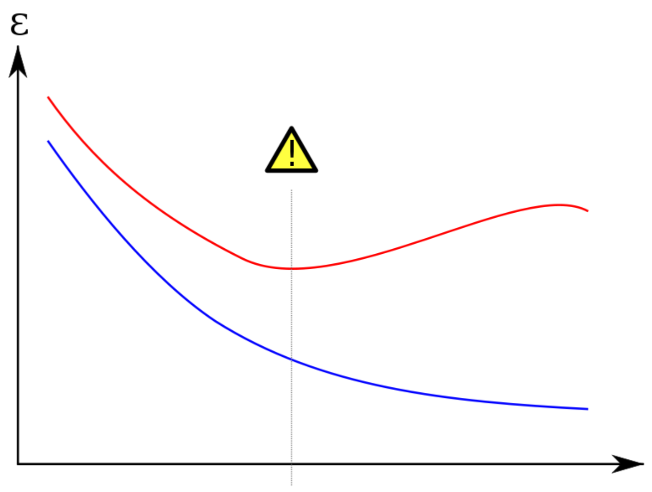


Figure 1.7: Challenges in Deep Learning (1) [18]

1.5.3 Hyperparameter Optimization:

Hyperparameters are the parameters whose value is defined prior to the commencement of the learning process. Changing the value of such parameters by a small amount can invoke a large change in the performance of your model.

Relying on the default parameters and not performing Hyperparameter Optimization can have a significant impact on the model performance. Also, having too few hyperparameters and hand tuning them rather than optimizing through proven methods is also a performance driving aspect.

1.5.4 Requires High Performance Hardware

Training a data set for a Deep Learning solution requires a lot of data. To perform a task to solve real world problems, the machine needs to be equipped with adequate processing power. To ensure better efficiency and less time consumption, data scientists switch to multi-core high performing GPUs and similar processing units. These processing units are costly and consume a lot of power.

Industry level Deep Learning systems require high-end data centers while smart devices such as drones, robots other mobile devices require small but efficient processing units. Deploying Deep Learning solution to the real world thus becomes a costly and power consuming affair.

1.5.5 Neural Networks are essentially a blackbox:

We know our model parameters, we feed known data to the neural networks and how they are put together. But we usually do not understand how they arrive at a particular solution. Neural networks are essentially Balckboxes and researchers have a hard time understanding how they deduce conclusions.

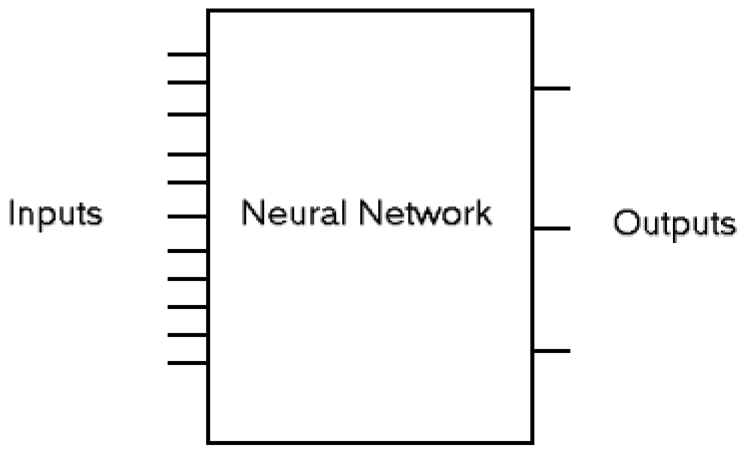


Figure 1.8: Challenges in Deep Learning (2) [18]

The lack of ability of neural networks for reason on an abstract level makes it difficult to implement high-level cognitive functions. Also, their operation is largely invisible to humans, rendering them unsuitable for domains in which verification of process is important.

Chapter 2

2 Literature Review

In this chapter, we first present a discussion on different geometric and appearance-based facial feature representation, which is followed by a review on different appearance-based methods. Finally, we end the literature review with the description of some pattern recognition methods used for the facial expression recognition system.

2.1 Convolutional Neural Network (CNN) Background

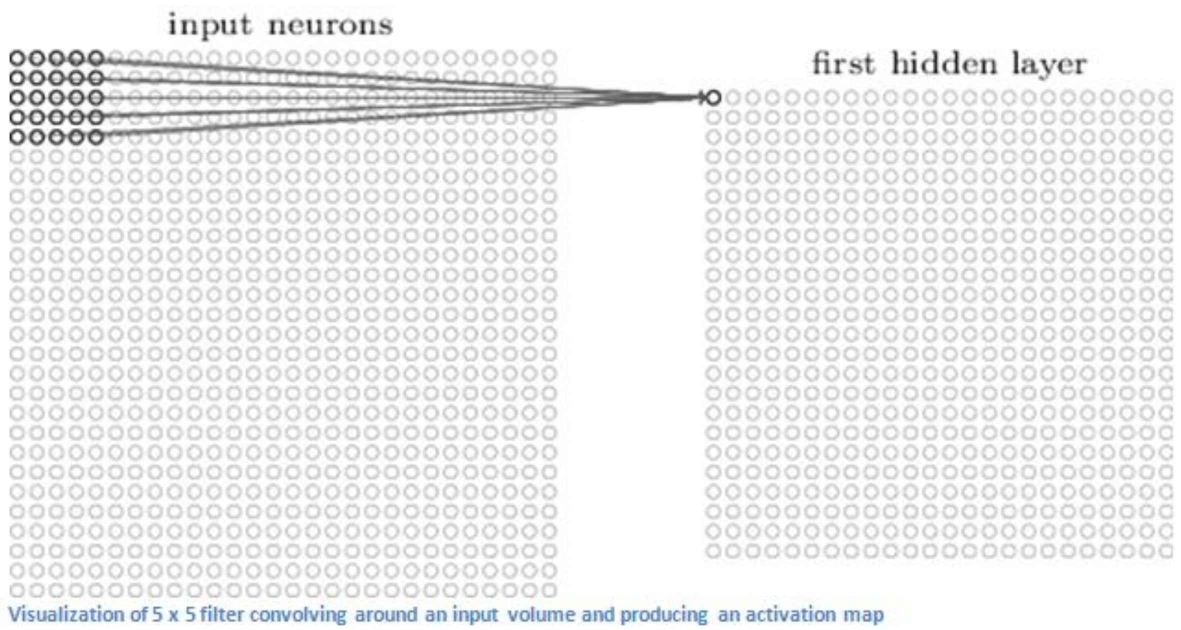
CNNs takes a biological inspiration from the visual cortex. The visual cortex has small regions of cells that are sensitive to specific regions of the visual field. This idea was expanded upon by a fascinating experiment by Hubel and Wiesel in 1962, where they showed that some individual neuronal cells in the brain responded (or fired) only in the presence of edges of a certain orientation.[8] For example, some neurons fired when exposed to vertical edges and some when shown horizontal or diagonal edges. Hubel and Wiesel found out that all of these neurons were organized in a columnar architecture and that together, they were able to produce visual perception. This idea of specialized components inside of a system having specific tasks (the neuronal cells in the visual cortex looking for specific characteristics) is one that machines use as well, and is the basis behind CNNs.

2.1.1 Structure

A more detailed overview of what CNNs do would be that we take the image, pass it through a series of convolutional, nonlinear, pooling (down sampling), and fully connected layers, and get an output. As we said earlier, the output can be a single class or a probability of classes that best describes the image. Now, the hard part is understanding what each of these layers do.

2.1.2 First Layer – Math Part

The first layer in a CNN is always a **Convolutional Layer**. First thing to make sure we remember is what the input to this convolution layer is. Like we mentioned before, the input is a 32 x 32 x 3 array of pixel values. Now, the best way to explain a convolution layer is to imagine a flashlight that is shining over the top left of the image. Let’s say that the light this flashlight shines covers a 5 x 5 area. And now, let’s imagine this flashlight sliding across all the areas of the input image. In machine learning terms, this flashlight is called a **filter** (or sometimes referred to as a **neuron** or a **kernel**) and the region that it is shining over is called the **receptive field**. Now this filter is also an array of numbers (the numbers are called **weights** or **parameters**). A very important note is that the depth of this filter has to be the same as the depth of the input (this makes sure that the math works out), so the dimensions of this filter is 5 x 5 x 3. Now, let’s take the first position the filter is in for example. It would be the top left corner. As the filter is sliding, or **convolving**, around the input image, it is multiplying the values in the filter with the original pixel values of the image (aka computing **element wise multiplications**). These multiplications are all summed up (mathematically speaking, this would be 75 multiplications in total). So now we have a single number. Remember, this number is just representative of when the filter is at the top left of the image. Now, we repeat this process for every location on the input volume. (Next step would be moving the filter to the right by 1 unit, then right again by 1, and so on). Every unique location on the input volume produces a number. After sliding the filter over all the locations, you will find out that what you’re left with is a 28 x 28 x 1 array of numbers, which we call an **activation map** or **feature map**. The reason you get a 28 x 28 array is that there are 784 different locations that a 5 x 5 filter can fit on a 32 x 32 input image. These 784 numbers are mapped to a 28 x 28 array.



Visualization of 5 x 5 filter convolving around an input volume and producing an activation map

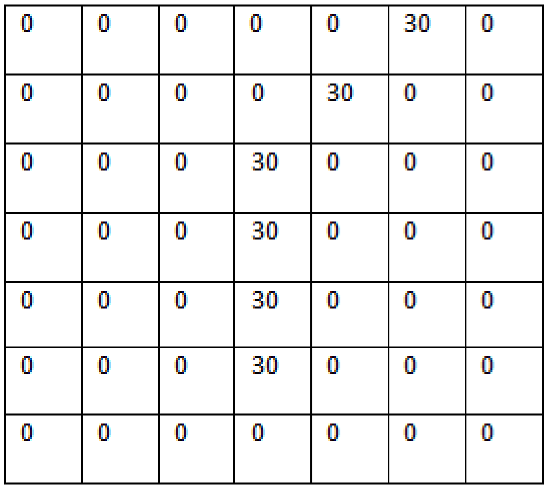
*Figure 2.1: 5*5 filter convolving around image for producing activation [19]*

Image from the book “Neural Networks and Deep learning” by Michael Nielsen. (Nielsen)

Let’s say now we use two 5 x 5 x 3 filters instead of one. Then our output volume would be 28 x 28 x 2. By using more filters, we are able to preserve the spatial dimensions better. Mathematically, this is what’s going on in a convolutional layer.

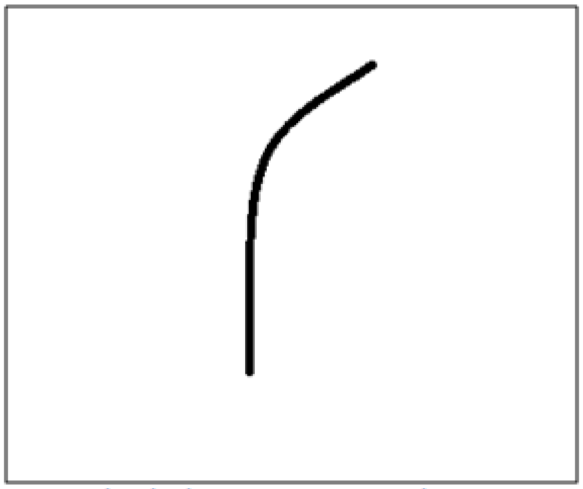
2.1.3 First Layer – High Level Perspective

Let’s talk about what this convolution is actually doing from a high level. Each of these filters can be thought of as **feature identifiers**. Here features means things like straight edges, simple colors, and curves. Think about the simplest characteristics that all images have in common with each other. Let’s say our first filter is 7 x 7 x 3 and is going to be a curve detector. (In this section, let’s ignore the fact that the filter is 3 units deep and only consider the top depth slice of the filter and the image, for simplicity.)As a curve detector, the filter will have a pixel structure in which there will be higher numerical values along the area that is a shape of a curve (Remember, these filters that we’re talking about as just numbers!).



0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter



Visualization of a curve detector filter

Figure 5: Visualizing activation after first layer [19]

Now, let’s go back to visualizing this mathematically. When we have this filter at the top left corner of the input volume, it is computing multiplications between the filter and pixel values at that region. Now let’s take an example of an image that we want to classify, and let’s put our filter at the top left corner.

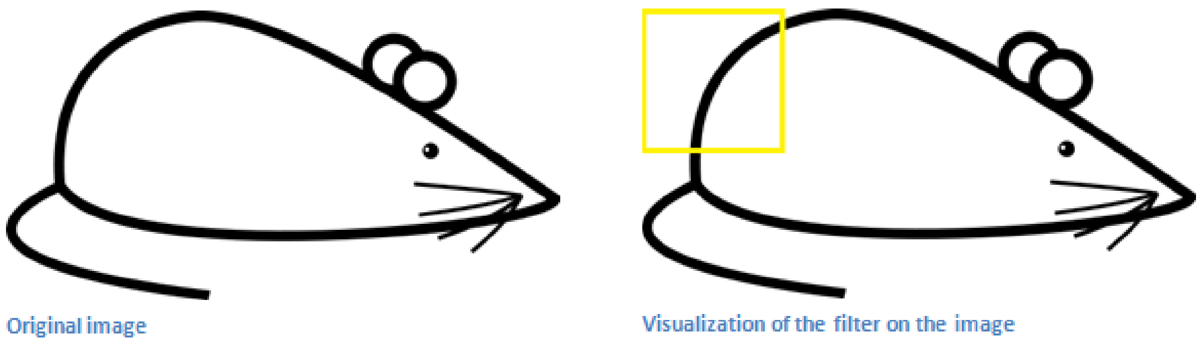
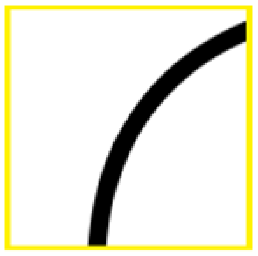


Figure 2.2: Visualizing activation in convolutional layer [19]

Remember, what we have to do is multiply the values in the filter with the original pixel values of the image.



0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation of the receptive field

*

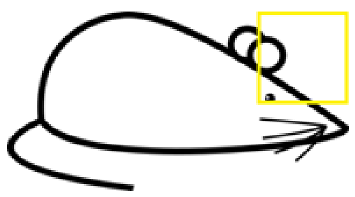
0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600 (A large number!)

Figure 2.3: Visualizing activation in convolutional layer (2) [19]

Basically, in the input image, if there is a shape that generally resembles the curve that this filter is representing, then all of the multiplications summed together will result in a large value! Now let’s see what happens when we move our filter.



0	0	0	0	0	0	0
0	40	0	0	0	0	0
40	0	40	0	0	0	0
40	20	0	0	0	0	0
0	50	0	0	0	0	0
0	0	50	0	0	0	0
25	25	0	50	0	0	0

Pixel representation of receptive field

$*$

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = 0

Figure 2.4: Visualizing activation in convolutional layer (3) [19]

The value is much lower! This is because there wasn't anything in the image section that responded to the curve detector filter. Remember, the output of this convolution layer is an activation map. So, in the simple case of a one filter convolution (and if that filter is a curve detector), the activation map will show the areas in which there at mostly likely to be curves in the picture. In this example, the top left value of our 28 x 28 x 1 activation map will be 6600. This high value means that it is likely that there is some sort of curve in the input volume that caused the filter to activate. The top right value in our activation map will be 0 because there wasn't anything in the input volume that caused the filter to activate (or more simply said, there wasn't a curve in that region of the original image). Remember, this is just for one filter. This is just a filter that is going to detect lines that curve outward and to the right. We can have other filters for lines that curve to the left or for straight edges. The more filters, the greater the depth of the activation map, and the more information we have about the input volume. [8]

2.2 Going Deeper Through the Network

Now in a traditional convolutional neural network architecture, there are other layers that are interspersed between these convolution layers. A classic CNN architecture would look like this. [9]

Input -> Conv -> ReLU -> Conv -> ReLU -> Pool -> ReLU -> Conv -> ReLU -> Pool ->Fully Connected

Let's just take a step back and review what we've learned so far. We talked about what the filters in the first conv layer are designed to detect. They detect low level features such as edges and curves. As one would imagine, in order to predict whether an image is a type of object, we need the network to be able to recognize higher level features such as hands or paws or ears. So let's think about what the output of the network is after the first conv layer. It would be a 28 x 28 x 3 volume (assuming we use three 5 x 5 x 3 filters). When we go through another convolution layer, the output of the first conv layer becomes the input of the 2nd convolution layer. Now, this is a little bit harder to visualize. When we were talking about the first layer, the input was just the original image. However, when we're talking about the 2nd conv layer, the input is the activation map(s) that result from the first layer. So each layer of the input is basically describing the locations in the original image for where certain low level features appear. Now when we apply a set of filters on top of that (pass it through the 2nd convolution layer), the output will be activations that represent higher level features. Types of these features could be semicircles (combination of a curve and straight edge) or squares (combination of several straight edges). As we go through the network and go through more convolution layers, we get activation maps that represent more and more complex features. By the end of the network, we may have some filters that activate when there is handwriting in the image, filters that activate when they see pink objects, etc. More information about visualizing filters in ConvNets, can be found in the paper written by Matt Zeiler and Rob Fergus. (Visualizing and Understanding Convolutional Neural Networks, 2014)

2.2.1 Fully Connected Layer

Now that we can detect these high level features, the icing on the cake is attaching a **fully connected layer** to the end of the network. This layer basically takes an input volume (whatever the output is of the convolution layer or ReLU or pool layer preceding it) and outputs an N dimensional vector where N is the number of classes that the program has to choose from. For example, if you wanted a digit classification program, N would be 10 since there are 10 digits. Each number in this N dimensional vector represents the probability of a certain class. For example, if the resulting vector for a digit classification program is [0 .1 .1 .75 0 0 0 0 0 .05], then this represents a 10% probability that the image is a 1, a 10% probability that the image is a 2, a 75% probability that the image is a 3, and a 5% probability that the image is a 9. The way this fully connected layer works is that it looks at the output of the previous layer (which as we remember should represent the activation maps of high level features) and determines which features most correlate to a particular class. For example, if the program is predicting that some image is a dog, it will have high values in the activation maps that represent high level features like a paw or 4 legs, etc. Similarly, if the program is predicting that some image is a bird, it will have high values in the activation maps that represent high level features like wings or a beak, etc. Basically, a FC layer looks at what high level features most strongly correlate to a particular class and has particular weights so that when you compute the products between the weights and the previous layer, you get the correct probabilities for the different classes.

2.2.2 Training

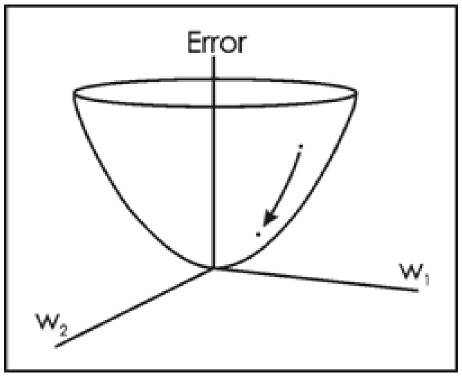
How do the filters in the first conv layer know to look for edges and curves? How does the fully connected layer know what activation maps to look at? How do the filters in each layer know what values to have? The way the computer is able to adjust its filter values (or weights) is through a training process called **backpropagation**.

Before we get into backpropagation, we must first take a step back and talk about what a neural network needs in order to work. At the moment we all were born, our minds were fresh. We didn’t know what a cat or dog or bird was. In a similar sort of way, before the CNN starts, the weights or filter values are randomized. The filters don’t know to look for edges and curves. The filters in the higher layers don’t know to look for paws and beaks. As we grew older however, our parents and teachers showed us different pictures and images and gave us a corresponding label. This idea of being given an image and a label is the training process that CNNs go through. Before getting too into it, let’s just say that we have a training set that has thousands of images of dogs, cats, and birds and each of the images has a label of what animal that picture is containing. Now, getting back to backpropagation.

So backpropagation can be separated into 4 distinct sections, the forward pass, the loss function, the backward pass, and the weight update. During the **forward pass**, we take a training image which as we remember is a 32 x 32 x 3 array of numbers and pass it through the whole network. On our first training example, since all of the weights or filter values were randomly initialized, the output will probably be something like [.1 .1 .1 .1 .1 .1 .1 .1 .1 .1], basically an output that doesn’t give preference to any number in particular. The network, with its current weights, isn’t able to look for those low level features or thus isn’t able to make any reasonable conclusion about what the classification might be. This goes to the **loss function** part of backpropagation. Remember that what we are using right now is training data. This data has both an image and a label. Let’s say for example that the first training image inputted was a 3. The label for the image would be [0 0 0 1 0 0 0 0 0 0]. A loss function can be defined in many different ways but a common one is MSE (mean squared error), which is ½ times (actual - predicted) squared.

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

Let’s say the variable L is equal to that value. As you can imagine, the loss will be extremely high for the first couple of training images. Now, let’s just think about this intuitively. We want to get to a point where the predicted label (output of the ConvNets) is the same as the training label (This means that our network got its prediction right).In order to get there, we want to minimize the amount of loss we have. Visualizing this as just an optimization problem in calculus, we want to find out which inputs (weights in our case) most directly contributed to the loss (or error) of the network.



One way of visualizing this idea of minimizing the loss is to consider a 3-D graph where the weights of the neural net (there are obviously more than 2 weights, but let's go for simplicity) are the independent variables and the dependent variable is the loss. The task of minimizing the loss involves trying to adjust the weights so that the loss decreases. In visual terms, we want to get to the lowest point in our bowl shaped object. To do this, we have to take a derivative of the loss (visual terms: calculate the slope in every direction) with respect to the weights.

Figure 2.5: Visualizing Error [19]

This is the mathematical equivalent of a **dL/dW** where we are the weights at a particular layer. Now, what we want to do is perform a **backward pass** through the network, which is determining which weights contributed most to the loss and finding ways to adjust them so that the loss decreases. Once we compute this derivative, we then go to the last step which is the **weight update**. This is where we take all the weights of the filters and update them so that they change in the opposite direction of the gradient.

$$w = w_i - \eta \frac{dL}{dW}$$

w = Weight
w_i = Initial Weight
η = Learning Rate

The **learning rate** is a parameter that is chosen by the programmer. A high learning rate means that bigger steps are taken in the weight updates and thus, it may take less time for the model to converge on an optimal set of weights. However, a learning rate that is too high could result in jumps that are too large and not precise enough to reach the optimal point.

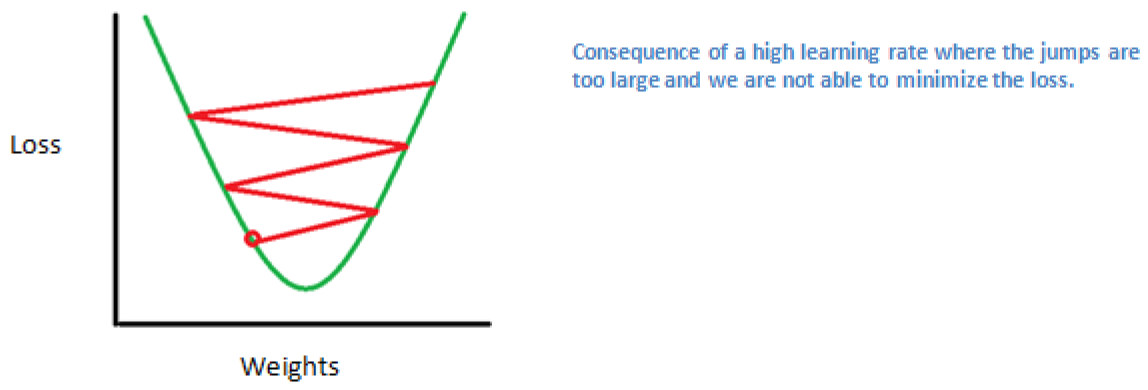

Loss
Weights
Consequence of a high learning rate where the jumps are too large and we are not able to minimize the loss.

Figure 2.6: Learning rate [19]

The process of forward pass, loss function, backward pass, and parameter update is one training iteration. The program will repeat this process for a fixed number of iterations for each set of training images (commonly called a batch). Once you finish the parameter update on the last training example, hopefully the network should be trained well enough so that the weights of the layers are tuned correctly.

2.2.3 Testing

Finally, to see whether or not our CNN works, we have a different set of images and labels (can't double dip between training and test!) and pass the images through the CNN. We compare the outputs to the ground truth and see if our network works.

2.2.4 Stride and Padding

There are 2 main parameters that we can change to modify the behavior of each layer. After we choose the filter size, we also have to choose the **stride** and the **padding**.

Stride controls how the filter convolves around the input volume. In the example previously given, the filter convolves around the input volume by shifting one unit at a time. The amount by which the filter shifts is the stride. In that case, the stride was implicitly set at 1. Stride is normally set in a way so that the output volume is an integer and not a fraction.

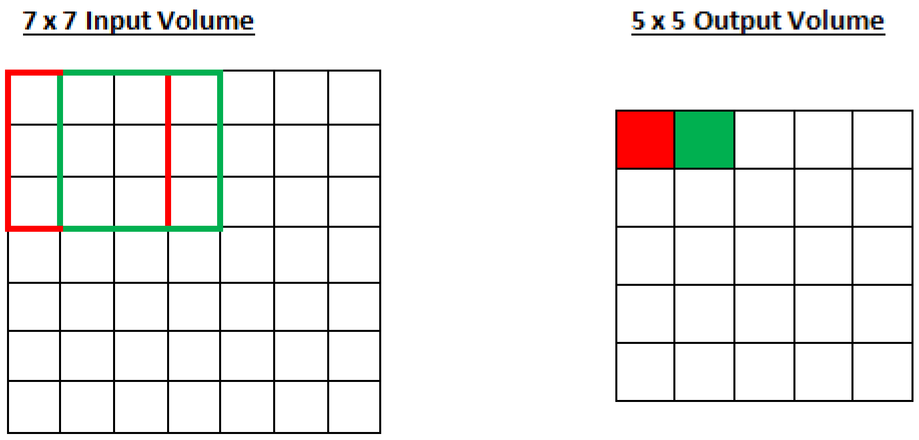
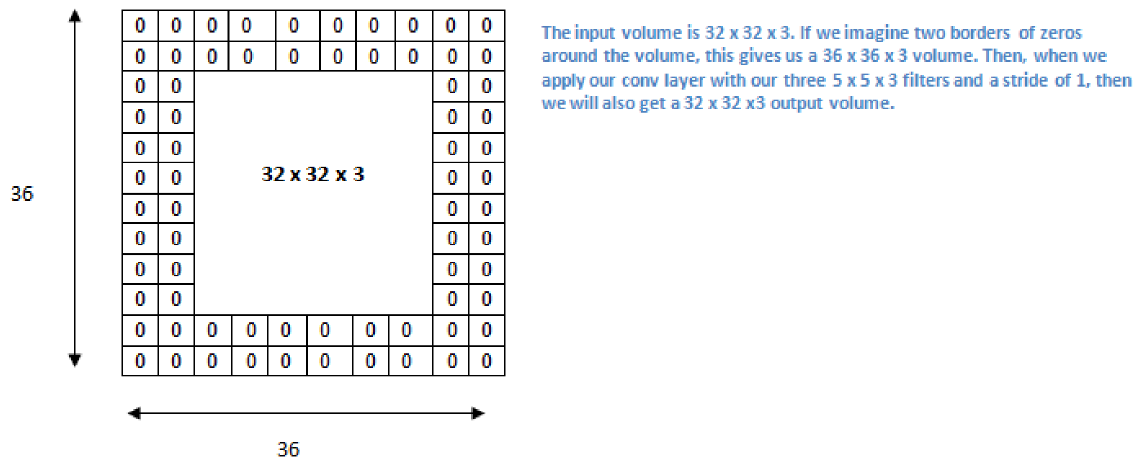


Figure 2.7: Stride and padding [19]

Now, let's take a look at padding. Before getting into that, let's think about a scenario. What happens when you apply three 5 x 5 x 3 filters to a 32 x 32 x 3 input volume? The output volume would be 28 x 28 x 3. Notice that the spatial dimensions decrease. As we keep applying conv layers, the size of the volume will decrease faster than we would like. In the early layers of our network, we want to preserve as much information about the original input volume so that we can extract those low level features. Let's say we want to apply the same conv layer but we want the output volume to remain 32 x 32 x 3. To do this, we can apply a zero padding of size 2 to that layer. Zero padding pads the input volume with zeros around the border. If we think about a zero padding of two, then this would result in a 36 x 36 x 3 input volume.



The input volume is 32 x 32 x 3. If we imagine two borders of zeros around the volume, this gives us a 36 x 36 x 3 volume. Then, when we apply our conv layer with our three 5 x 5 x 3 filters and a stride of 1, then we will also get a 32 x 32 x3 output volume.

Figure 2.8: Stride and padding (2) [19]

If you have a stride of 1 and if you set the size of zero padding to

$$Zero\ Padding = \frac{(K-1)}{2}$$

Where K is the filter size, then the input and output volume will always have the same spatial dimensions.

The formula for calculating the output size for any given convolution layer is

$$O = \frac{(W - K + 2P)}{S} + 1$$

Where O is the output height/length, W is the input height/length, K is the filter size, P is the padding, and S is the stride.

2.2.5 Choosing hyper parameters

How do we know how many layers to use, how many conv layers, what are the filter sizes, or the values for stride and padding? These are not trivial questions and there isn't a set standard that is used by all researchers. This is because the network will largely depend on the type of data that we have. Data can vary by size, complexity of the image, type of image processing task, and more. When looking at our dataset, one way to think about how to choose the hyperparameters is to find the right combination that creates abstractions of the image at a proper scale.

2.2.6 ReLU (Rectified Linear Units) Layers

After each conv layer, it is convention to apply a nonlinear layer (or **activation layer**) immediately afterward. The purpose of this layer is to introduce nonlinearity to a system that basically has just been computing linear operations during the conv layers (just element wise multiplications and summations).In the past, nonlinear functions like tanh and sigmoid were used, but researchers found out that **ReLU layers** work far better because the network is able to train a lot faster (because of the computational efficiency) without making a significant difference to the accuracy. It also helps to alleviate the vanishing gradient problem, which is the issue where the lower layers of the network train very slowly because the gradient decreases exponentially through the layers. The ReLU layer applies the function f(x) = max (0, x) to all of the values in the input volume. In basic terms, this layer just changes all the negative activations to 0.This layer increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the convolution layer. [10]

A paper is present related to ReLU by Geoffrey Hinton. (Rectified Linear Units Improve Restricted Boltzmann Machines, 2014)

2.2.7 Pooling Layers

After some ReLU layers, programmers may choose to apply a **pooling layer**. It is also referred to as a downsampling layer. In this category, there are also several layer options, with maxpooling being the most popular. This basically takes a filter (normally of size 2x2) and a stride of the same length. It then applies it to the input volume and outputs the maximum number in every sub region that the filter convolves around. [11]

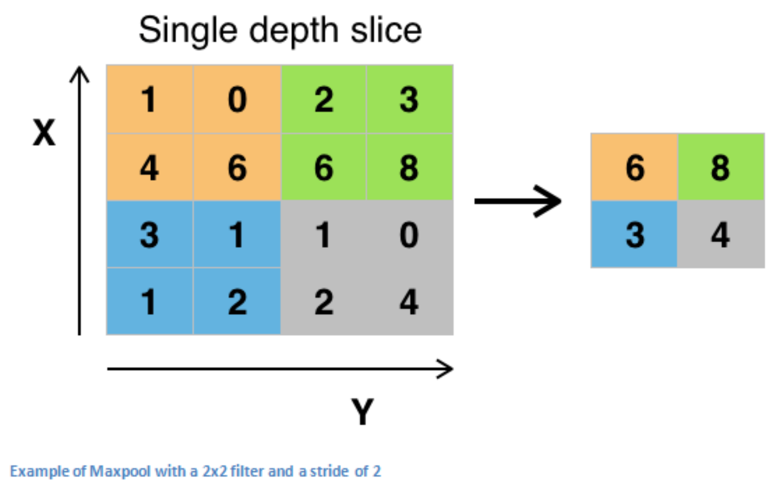


Figure 2.9: Pooling layer [19]

Other options for pooling layers are average pooling and L2-norm pooling. The intuitive reasoning behind this layer is that once we know that a specific feature is in the original input volume (there will be a high activation value), its exact location is not as important as its relative location to the other features. As you can imagine, this layer drastically reduces the spatial dimension (the length and the width change but not the depth) of the input volume. This serves two main purposes. The first is that the amount of parameters or weights is reduced by 75%, thus lessening the computation cost. The second is that it will control **overfitting**. This term refers to when a model is so tuned to the training examples that it is not able to generalize well for the validation and test sets. A symptom of overfitting is having a model that gets 100% or 99% on the training set, but only 50% on the test data.

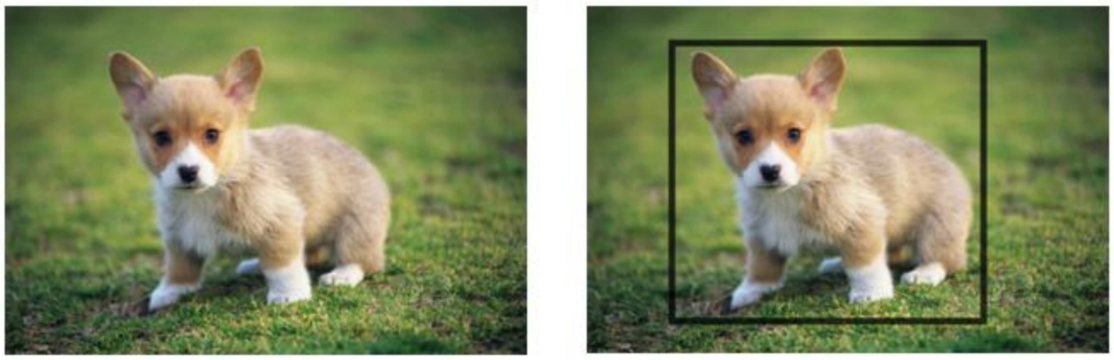
2.2.8 Dropout Layers

The idea of dropout is simplistic in nature. This layer "drops out" a random set of activations in that layer by setting them to zero in the forward pass. Simple as that. Now, what are the benefits of such a simple and seemingly unnecessary and counterintuitive process? Well, in a way, it forces the network to be redundant. By that we mean the network should be able to provide the right classification or output for a specific example even if some of the activations are dropped out. It makes sure that the network isn't getting too "fitted" to the training data and thus helps alleviate the overfitting problem. An important note is that this layer is only used during training, and not during test time.

Another paper by Geoffrey Hinton discusses on the subject matter. (Dropout: A Simple Way to Prevent Neural Networks from Overfitting, 2014)

2.3 Classification, Localization, Detection, Segmentation

In the example mentioned previously we looked at the task of **image classification**. This is the process of taking an input image and outputting a class number out of a set of categories. However, when we take a task like **object localization**, our job is not only to produce a class label but also a bounding box that describes where the object is in the picture.

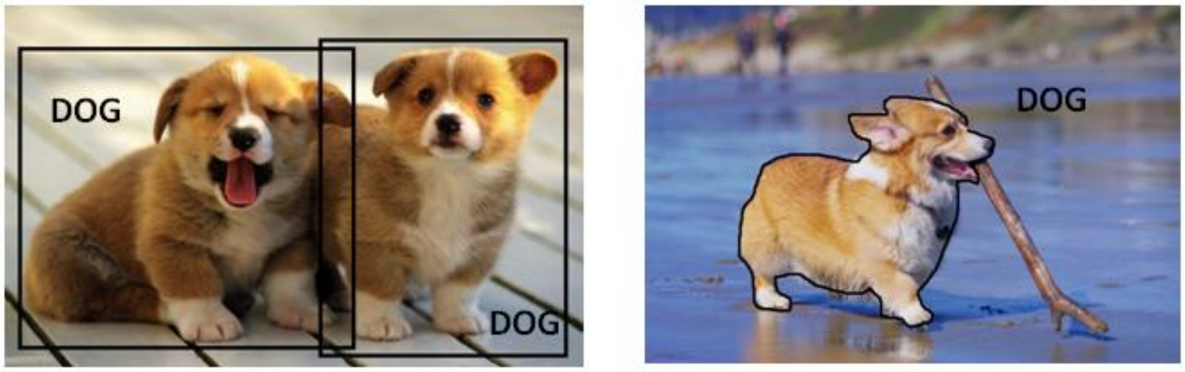


Object Classification is the task of identifying that picture is a dog

Object Localization involves the class label as well as a bounding box to show where the object is located.

Figure 2.10: Object classification and Localization

We also have the task of **object detection**, where localization needs to be done on all of the objects in the image. Therefore, you will have multiple bounding boxes and multiple class labels. Finally, we also have **object segmentation** where the task is to output a class label as well as an outline of every object in the input image. [11]



Object Detection involves localization of multiple objects (doesn't have to be the same class).

Object Segmentation involves the class label as well as an outline of the object in interest.

Figure 2.11: Object Detection and Segmentation

2.4 Transfer Learning

Now, a common misconception in the DL community is that without a Google-esque amount of data, you can't possibly hope to create effective deep learning models. While data is a critical part of creating the network, the idea of transfer learning has helped to lessen the data demands. **Transfer learning** is the process of taking a pre-trained model (the weights and parameters of a network that has been trained on a large dataset by somebody else) and "fine-tuning" the model with our own dataset. The idea is that this pre-trained model will act as a feature extractor. We will remove the last layer of the network and replace it with your own classifier (depending on what your problem space is). We then freeze the weights of all the other layers and train the network normally (Freezing the layers means not changing the weights during gradient descent/optimization).

2.5 Data Augmentation Techniques

By now, we're all probably numb to the importance of data in ConvNets, so let's talk about ways that you can make your existing dataset even larger, just with a couple easy transformations. Like we've mentioned before, when a computer takes an image as an input, it will take in an array of pixel values. Let's say that the whole image is shifted left by 1 pixel. To us, this change is imperceptible. However, to a computer, this shift can be fairly significant as the classification or label of the image doesn't change, while the array does. Approaches that alter the training data in ways that change the array representation while keeping the label the same are known as **data augmentation** techniques. They are a way to artificially expand your dataset. Some popular augmentations people use are grayscales, horizontal flips, vertical flips, random crops, color jitters, translations, rotations, and much more. By applying just a couple of these transformations to your training data, you can easily double or triple the number of training examples.

2.6 ImageNet Classification

This paper, titled "ImageNet Classification with Deep Convolutional Networks", has been cited a total of 6,184 times and is widely regarded as one of the most influential publications in the field. Alex Krizhevsky, IlyaSutskever, and Geoffrey Hinton created a "large, deep convolutional neural network" that was used to win the 2012 ILSVRC (ImageNet Large-Scale Visual Recognition Challenge). For those that aren't familiar, this competition can be thought of as the annual Olympics of computer vision, where teams from across the world compete to see who has the best computer vision model for tasks such as classification, localization, detection, and more. 2012 marked the first year where a CNN was used to achieve a top 5 test error rate of 15.4% (Top 5 error is the rate at which, given an image, the model does not output the correct label with its top 5 predictions). The next best entry achieved an error of 26.2%, which was an astounding improvement that pretty much shocked the computer vision community. Safe to say, CNNs became household names in the competition from then on out.

In the paper, the group discussed the architecture of the network (which was called AlexNet). They used a relatively simple layout, compared to modern architectures. The network was made up of 5 conv layers, max-pooling layers, dropout layers, and 3 fully connected layers. The network they designed was used for classification with 1000 possible categories.

Main Points:

- Trained the network on ImageNet data, which contained over 15 million annotated images from a total of over 22,000 categories.

- Used ReLU for the nonlinearity functions (Found to decrease training time as ReLUs are several times faster than the conventional tanh function).
- Used data augmentation techniques that consisted of image translations, horizontal reflections, and patch extractions.
- Implemented dropout layers in order to combat the problem of overfitting to the training data.
- Trained the model using batch stochastic gradient descent, with specific values for momentum and weight decay.
- Trained on two GTX 580 GPUs for **five to six days**.

The neural network developed by Krizhevsky, Sutskever, and Hinton in 2012 was the coming out party for CNNs in the computer vision community. This was the first time a model performed so well on a historically difficult ImageNet dataset. Utilizing techniques that are still used today, such as data augmentation and dropout, this paper really illustrated the benefits of CNNs and backed them up with record breaking performance in the competition. [11]

2.7 Visualizing Convolutional Neural Networks

In this paper titled "Visualizing and Understanding Convolutional Neural Networks", Zeiler and Fergus[12] begin by discussing the idea that this renewed interest in CNNs is due to the accessibility of large training sets and increased computational power with the usage of GPUs. They also talk about the limited knowledge that researchers had on inner mechanisms of these models, saying that without this insight, the "development of better models is reduced to trial and error". While we do currently have a better understanding than 3 years ago, this still remains an issue for a lot of researchers! The main contributions of this paper are details of a slightly modified AlexNet model and a very interesting way of visualizing feature maps.

Main Points:

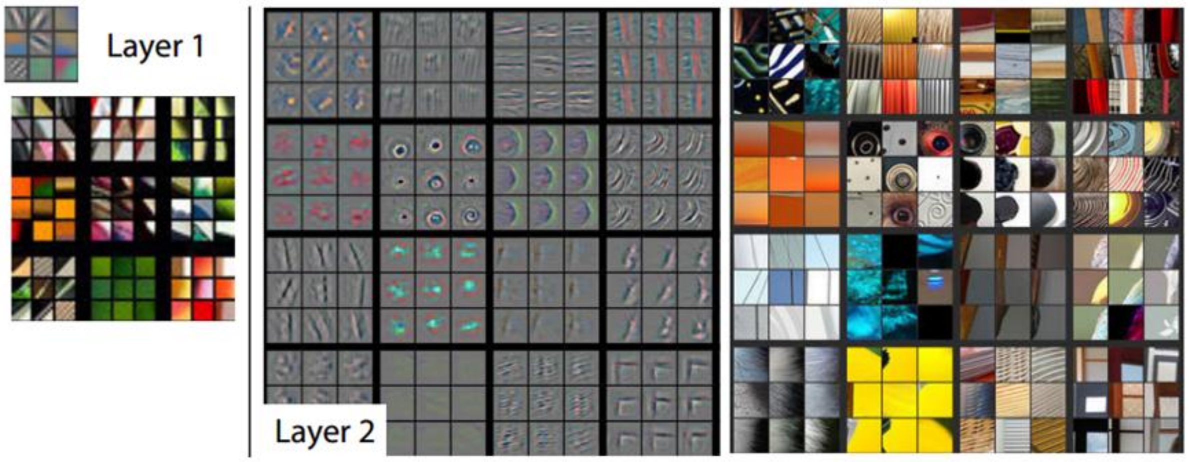
- Very similar architecture to AlexNet, except for a few minor modifications.
- AlexNet trained on 15 million images, while ZF Net trained on only 1.3 million images.
- Instead of using 11x11 sized filters in the first layer (which is what AlexNet implemented), ZF Net used filters of size 7x7 and a decreased stride value. The reasoning behind this modification is that a smaller filter size in the first conv layer helps retain a lot of original pixel information in the input volume. A filtering of size 11x11 proved to be skipping a lot of relevant information, especially as this is the first conv layer.
- As the network grows, we also see a rise in the number of filters used.
- Used ReLUs for their activation functions, cross-entropy loss for the error function, and trained using batch stochastic gradient descent.
- Trained on a GTX 580 GPU for **twelve days**.
- Developed a visualization technique named Deconvolutional Network, which helps to examine different feature activations and their relation to the input space. Called "deconvnet" because it maps features to pixels (the opposite of what a convolutional layer does).

2.7.1 DeConvNet

The basic idea behind how this works is that at every layer of the trained CNN, you attach a "deconvnet" which has a path back to the image pixels. An input image is fed into the CNN and activations are computed at each level. This is the forward pass. Now, let's say we want to examine the activations of a certain feature in the 4th conv layer. We would store the activations of this one feature map, but set all of the other activations in the layer to 0, and then pass this feature map as the input into the deconvnet. This deconvnet has the same filters as the original CNN. This input then goes through a series of unpool (reverse maxpooling), rectify, and filter operations for each preceding layer until the input space is reached. [12]

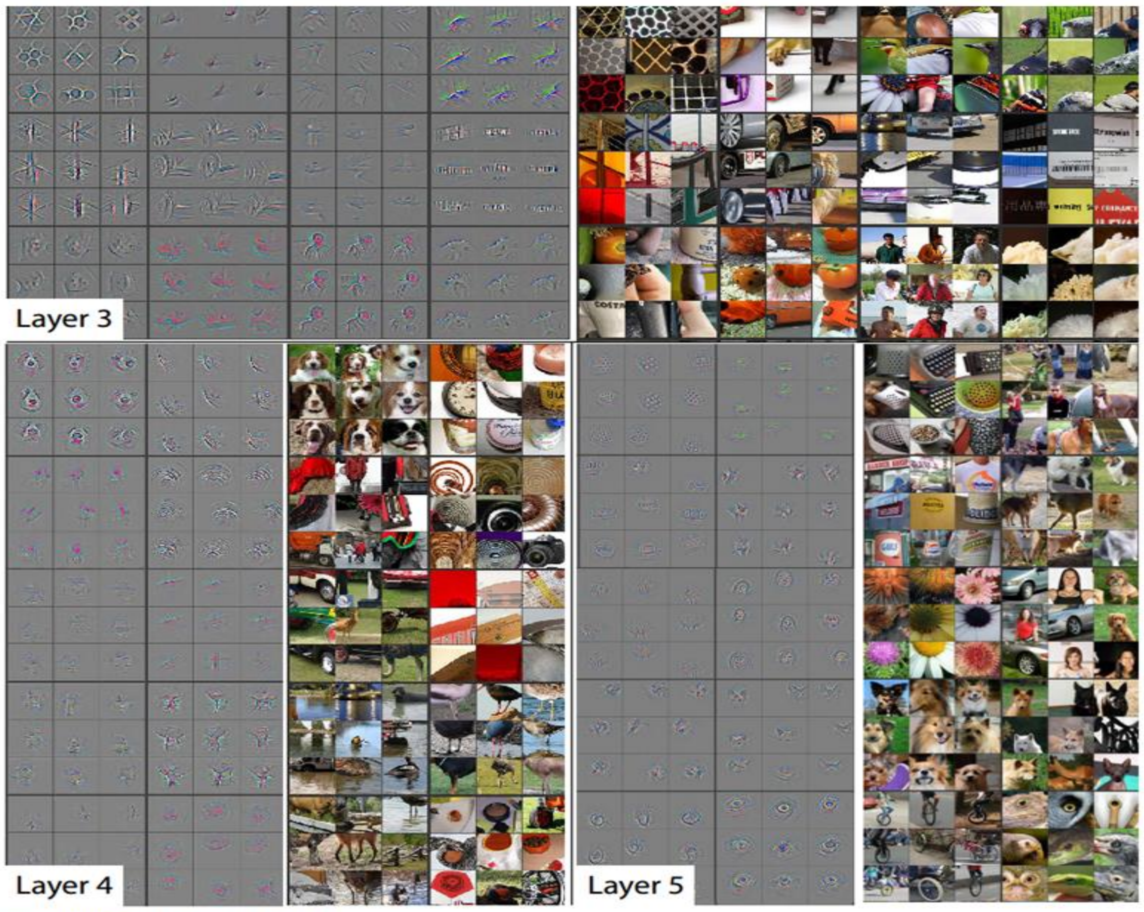
The reasoning behind this whole process is that we want to examine what type of structures excite a given feature map. Let's look at the visualizations of the first and second layers.

ZF Net was not only the winner of the competition in 2013, but also provided great intuition as to the workings on CNNs and illustrated more ways to improve performance. The visualization approach described helps not only to explain the inner workings of CNNs, but also provides insight for improvements to network architectures.



Visualizations of Layer 1 and 2. Each layer illustrates 2 pictures, one which shows the filters themselves and one that shows what part of the image are most strongly activated by the given filter. For example, in the space labled Layer 2, we have representations of the 16 different filters (on the left)

Figure 2.12: DeConvolutional Network [12]



Visualizations of Layers 3, 4, and 5

Figure 2.13: Deconvolutional Network (2) [12]

Chapter 3

3 Deep learning frameworks

Currently a limited variety of tools are available in terms of deep learning frameworks since they implement algorithms which are used in bleeding edge applications such as computer vision and machine translation. It is necessary to select the proper framework for proper modelling of deep neural networks. Following section discusses the overview of deep learning and open source frameworks such as TensorFlow, CNTK, Theano, Torch, Caffe, MXnet and Neon.

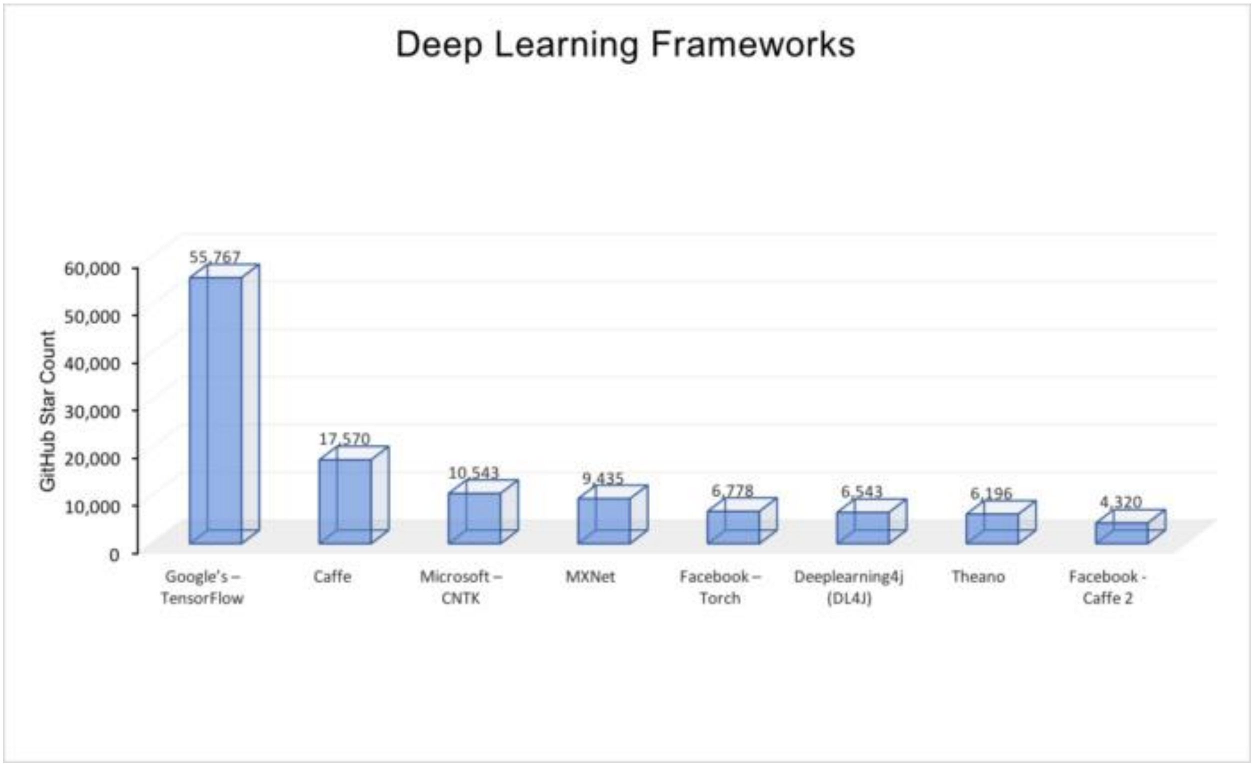


Figure 3.1: GitHub star count of different deep learning frameworks [20]

3.1 Theano:

Developed at the LISA lab at the University of Montreal, Theano consists of a Python library which allows users to define, optimize, and evaluate mathematical expressions on arrays and tensors. It also generates customized C code for various mathematical operations. In terms of architecture, when compared to other deep learning frameworks available the popular consensus is that Theano is difficult to use. Theano also has multiple GPU support and provides support for convolutional neural networks and recurrent neural networks. Theano has been deployed in production in AI services companies such as Parallel Dots serving up to a few 1000 concurrent news recommendations and multiple client Deep Learning as a service calls.

3.2 TensorFlow

TensorFlow was originally developed by researchers and engineers whom worked on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research and over time the framework has been found to be general enough to be applicable in a wide variety of other domains as well. [**Deng&Yu**] TensorFlow is implemented in Python and uses data flow graphs for numerical computation. When compared to other deep learning frameworks it can be seen that TensorFlow is considered to currently be the best documented open source framework available TensorFlow also boasts of an easy to use and modular front-end in terms of architecture. TensorFlow also provides supports for Convolutional Neural Networks and Recurrent Neural Networks. It should also be noted that there are implementations for Restricted Boltzmann Machines, Deep Autoencoders and Long Short-Term models utilizing TensorFlow architecture as well. In addition, TensorFlow also contains TensorBoard which is a suite of visualization tools which make it easier to understand, debug and optimize programs which run on TensorFlow code.

3.3 Torch:

Torch is a deep learning framework with support for algorithms that give priority to GPUs. Torch provides faster performance compared to other deep learning frameworks due to the use of the fast scripting language LuaJIT and its underlying C/CUDA implementation. Torch also possesses a large ecosystem of community driven packages and is also embeddable with iOS and Android backends. Torch also possesses Recurrent Neural Network and Convolution Neural Network modelling capability with a relatively easy to use modular front end. Torch is in use in companies such as Facebook, Google and Twitter as well as across research labs such as NYU, IDIAP and Purdue.

3.4 Caffe:

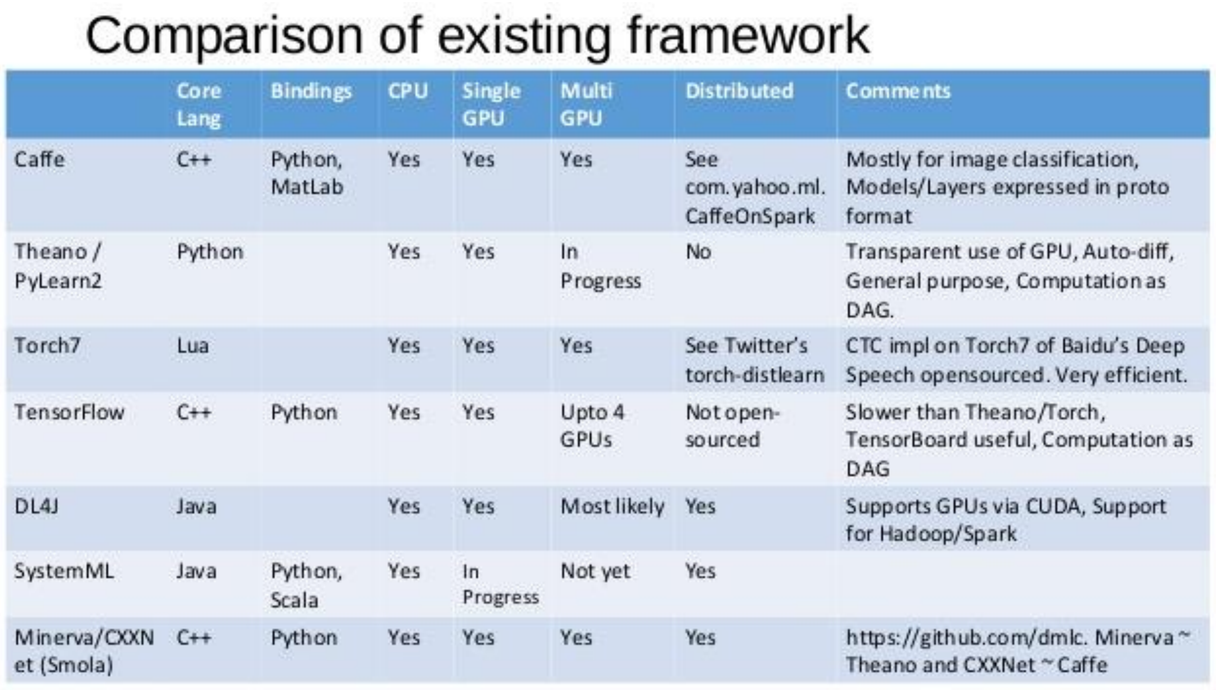
Caffe is a deep learning framework developed by Berkeley AI Research (BAIR) as well as community collaborators. It is implemented in C++. Caffe is released under the BSD 2- Clause license. [**caff01**] While Caffe supports Convolutional Neural Networks, it does not currently support Recurrent Neural Networks. Furthermore, even after extensive review of literature it was not possible to find any reference of many major players in the AI space deploying Caffe in a production environment. However Facebook recently released Caffe2 in April, which is a production ready cross-platform network and has been declared the successor to Caffe.

3.5 MXNet:

MXNet is a deep learning framework developed by collaborators from various companies and universities including the likes of Microsoft, Nvidia, Baidu, Intel, Carnegie Mellon University, University of Alberta and University of Washington. [**mxne01**] MXNet supports a plethora of programming languages including R, Python, Julia and Scala. It also has advanced GPU support compared to the other frameworks and also is relatively fast with regard to run time of deep learning algorithms. MXNet also has Convolutional Neural Network and Recurrent Neural Network modelling capabilities as well. MXNet is also Amazon Web Services (AWS)'s deep learning framework of choice.

3.6 NEON:

Neon is Intel Nervana's reference deep learning framework which has been designed for extensibility and ease of use. [**neon01**] Neon supports Python and supports deep learning models such as Convolutional Neural Networks, Recurrent Neural Networks, Long Short-Term Memory (LSTM) Models and Deep Autoencoders. Furthermore, Neon is also tightly integrated with Intel's GPU kernel library.



Comparison of existing framework

	Core Lang	Bindings	CPU	Single GPU	Multi GPU	Distributed	Comments
Caffe	C++	Python, MatLab	Yes	Yes	Yes	See com.yahoo.ml. CaffeOnSpark	Mostly for image classification, Models/Layers expressed in proto format
Theano / PyLearn2	Python		Yes	Yes	In Progress	No	Transparent use of GPU, Auto-diff, General purpose, Computation as DAG.
Torch7	Lua		Yes	Yes	Yes	See Twitter's torch-distlearn	CTC impl on Torch7 of Baidu's Deep Speech opensourced. Very efficient.
TensorFlow	C++	Python	Yes	Yes	Upto 4 GPUs	Not open-sourced	Slower than Theano/Torch, TensorBoard useful, Computation as DAG
DL4J	Java		Yes	Yes	Most likely	Yes	Supports GPUs via CUDA, Support for Hadoop/Spark
SystemML	Java	Python, Scala	Yes	In Progress	Not yet	Yes	
Minerva/CXXNet (Smola)	C++	Python	Yes	Yes	Yes	Yes	https://github.com/dmlc. Minerva ~ Theano and CXXNet ~ Caffe

Figure 3.2: Comparison of existing frameworks [20]

Chapter 4

4 Experimental Analysis

4.1 Implementer Systems:

4.1.1 TensorFlow

TensorFlow is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. TensorFlow was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well.

TensorFlow relies on a highly efficient C++ backend to do its computation. The connection to this backend is called a session. The common usage for TensorFlow programs is to first create a graph and then launch it in a session.

Here we instead use the convenient Interactive Session class, which makes TensorFlow more flexible about how you structure your code. It allows you to interleave operations which build a computation graph with ones that run the graph. This is particularly convenient when working in interactive contexts like IPython. If you are not using an InteractiveSession, then you should build the entire computation graph before starting a session and launching the graph.

4.1.1.1 Computation Graph

To do efficient numerical computing in Python, we typically use libraries like NumPy that do expensive operations such as matrix multiplication outside Python, using highly efficient code implemented in another language. Unfortunately, there can still be a lot of overhead from switching back to Python every operation. This overhead is especially bad if you want to run computations on GPUs or in a distributed manner, where there can be a high cost to transferring data.

TensorFlow also does its heavy lifting outside Python, but it takes things a step further to avoid this overhead. Instead of running a single expensive operation independently from Python, TensorFlow lets us describe a graph of interacting operations that run entirely outside Python. This approach is similar to that used in Theano or Torch.

The role of the Python code is therefore to build this external computation graph, and to dictate which parts of the computation graph should be run.

4.1.1.2 Placeholders

We start building the computation graph by creating nodes for the input images and target output classes. Here `x` and `y_` aren't specific values. Rather, they are each a `placeholder` -- a value that we'll input when we ask TensorFlow to run a computation.

The input images `x` will consist of a 2d tensor of floating point numbers. Here we assign it a `shape` of `[None, 784]`, where `784` is the dimensionality of a single flattened 28 by 28 pixel MNIST image, and `none` indicates that the first dimension, corresponding to the batch size, can be of any size. The target output classes `y_` will also consist of a 2d tensor, where each row is a one-hot 10-dimensional vector indicating which digit class (zero through nine) the corresponding MNIST image belongs to.

The `shape` argument to `placeholder` is optional, but it allows TensorFlow to automatically catch bugs stemming from inconsistent tensor shapes.

4.1.1.3 Variables

We now define the weights `W` and biases `b` for our model. We could imagine treating these like additional inputs, but TensorFlow has an even better way to handle them: `Variable`. A `Variable` is a value that lives in TensorFlow's computation graph. It can be used and even modified by the computation. In machine learning applications, one generally has the model parameters be `Variable`s.

We pass the initial value for each parameter in the call to `tf.Variable`. In this case, we initialize both `W` and `b`as tensors full of zeros. `W` is a 784x10 matrix (because we have 784 input features and 10 outputs) and `b` is a 10-dimensional vector (because we have 10 classes).

Before `Variable`s can be used within a session, they must be initialized using that session. This step takes the initial values (in this case tensors full of zeros) that have already been specified, and assigns them to each `Variable`.

4.1.1.4 Predicted Class and Loss Function

We can now implement our regression model. It only takes one line! We multiply the vectorized input images `x`by the weight matrix `W`, add the bias `b`.

We can specify a loss function just as easily. Loss indicates how bad the model's prediction was on a single example; we try to minimize that while training across all the examples. Here, our loss function is the cross-entropy between the target and the softmax activation function applied to the model's prediction.

Note that `tf.nn.softmax_cross_entropy_with_logits` internally applies the softmax on the model's unnormalized model prediction and sums across all classes, and `tf.reduce_mean` takes the average over these sums.

4.1.1.5 Train the Model

Now that we have defined our model and training loss function, it is straightforward to train using TensorFlow. Because TensorFlow knows the entire computation graph, it can use automatic differentiation to find the gradients of the loss with respect to each of the variables. TensorFlow has a variety of built-in optimization algorithms. For this example, we will use steepest gradient descent, with a step length of 0.5, to descend the cross entropy.

What TensorFlow actually did in that single line was to add new operations to the computation graph. These operations included ones to compute gradients, compute parameter update steps, and apply update steps to the parameters.

The returned operation `train_step`, when run, will apply the gradient descent updates to the parameters. Training the model can therefore be accomplished by repeatedly running `train_step`.

We load 100 training examples in each training iteration. We then run the `train_step` operation, using `feed_dict` to replace the `placeholder` tensors `x` and `y_` with the training examples. Note that you can replace any tensor in your computation graph using `feed_dict` -- it's not restricted to just `placeholder`s.

4.1.1.6 Evaluate the Model

First we'll figure out where we predicted the correct label. `tf.argmax` is an extremely useful function which gives you the index of the highest entry in a tensor along some axis. For example, `tf.argmax(y, 1)` is the label our model thinks is most likely for each input, while `tf.argmax (y_, 1)` is the true label. We can use `tf.equal` to check if our prediction matches the truth.

That gives us a list of Booleans. To determine what fraction are correct, we cast to floating point numbers and then take the mean. For example, `[True, False, True, True]` would become `[1, 0, 1, 1]` which would become `0.75`.

Finally, we can evaluate our accuracy on the test data. This should be about 92% correct.

4.1.1.7 Build a Multilayer Convolutional Network

Getting 92% accuracy on MNIST is bad. It's almost embarrassingly bad. In this section, we'll fix that, jumping from a very simple model to something moderately sophisticated: a small convolutional neural network. This will get us to around 99.2% accuracy -- not state of the art, but respectable.

Here is a diagram, created with TensorBoard, of the model we will build:

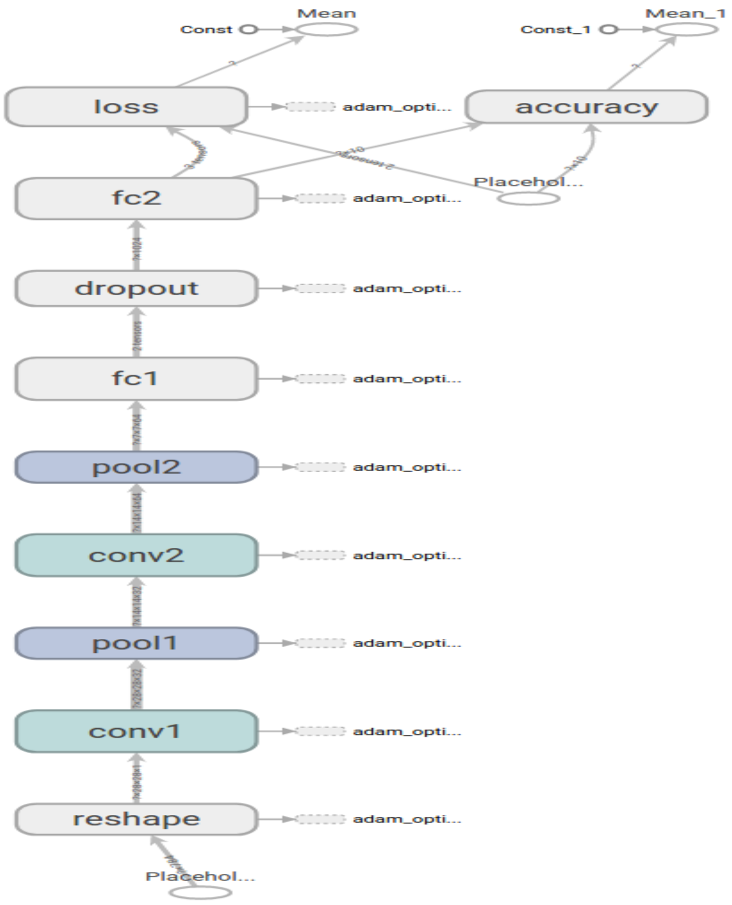


Figure 4.1: TensorFlow architecture [21]

4.1.1.8 Weight Initialization

To create this model, we're going to need to create a lot of weights and biases. One should generally initialize weights with a small amount of noise for symmetry breaking, and to prevent 0 gradients. Since we're using ReLU neurons, it is also good practice to initialize them with a slightly positive initial bias to avoid "dead neurons". Instead of doing this repeatedly while we build the model, let's create two handy functions to do it for us.

4.1.1.9 Convolution and Pooling

TensorFlow also gives us a lot of flexibility in convolution and pooling operations. How do we handle the boundaries? What is our stride size? In this example, we're always going to choose the vanilla version. Our convolutions uses a stride of one and are zero padded so that the output is the same size as the input. Our pooling is plain old max pooling over 2x2 blocks.

4.1.1.10 First Convolutional Layer

We can now implement our first layer. It will consist of convolution, followed by max pooling. The convolution will compute 32 features for each 5x5 patch. Its weight tensor will have a shape of [5, 5, 1, and 32]. The first two dimensions are the patch size, the next is the number of input channels, and the last is the number of output channels. We will also have a bias vector with a component for each output channel.

To apply the layer, we first reshape x to a 4d tensor, with the second and third dimensions corresponding to image width and height, and the final dimension corresponding to the number of color channels.

We then convolve x_image with the weight tensor, add the bias, apply the ReLU function, and finally max pool. The max_pool_2x2 method will reduce the image size to 14x14.

4.1.1.11 Second Convolutional Layer

In order to build a deep network, we stack several layers of this type. The second layer will have 64 features for each 5x5 patch.

4.1.1.12 Densely Connected Layer

Now that the image size has been reduced to 7x7, we add a fully-connected layer with 1024 neurons to allow processing on the entire image. We reshape the tensor from the pooling layer into a batch of vectors, multiply by a weight matrix, add a bias, and apply a ReLU.

4.1.1.13 Dropout

To reduce overfitting, we will apply dropout before the readout layer. We create a placeholder for the probability that a neuron's output is kept during dropout. This allows us to turn dropout on during training, and turn it off during testing. TensorFlow's tf.nn.dropout op automatically handles scaling neuron outputs in addition to masking them, so dropout just works without any additional scaling.[1]

4.1.1.14 Readout Layer

Finally, we add a layer, just like for the one layer softmax regression above.

4.1.1.15 Train and Evaluate the Model

To train and evaluate it we will use code that is nearly identical to that for the simple one layer SoftMax network above.

The differences are that:

- We will replace the steepest gradient descent optimizer with the more sophisticated ADAM optimizer.

- We will include the additional parameter keep_prob in feed_dict to control the dropout rate.
- We will add logging to every 100th iteration in the training process.

We will also use tf.Session rather than tf.InteractiveSession. This better separates the process of creating the graph (model specification) and the process of evaluating the graph (model fitting). It generally makes for cleaner code. The tf.Session is created within a with block so that it is automatically destroyed once the block is exited.

The final test set accuracy after running this code should be approximately 99.2%.

4.1.2 YOLO: Real Time Object Detection

You only look once (YOLO) is a state-of-the-art, real-time object detection system. On a Titan X it processes images at 40-90 FPS and has a mAP on VOC 2007 of 78.6% and a mAP of 48.1% on COCO test-dev. Prior detection systems repurpose classifiers or localizers to perform detection. They apply the model to an image at multiple locations and scales. High scoring regions of the image are considered detections.[14]

4.1.2.1 Description:

A single neural network is used to the full image. This network divides the image into regions and predicts bounding boxes and probabilities for each region. These bounding boxes are weighted by the predicted probabilities.

Compared to other region proposal classification networks (fast RCNN) which perform detection on various region proposals and thus end up performing prediction multiple times for various regions in an image, Yolo architecture is more like FCNN (fully convolutional neural network) and passes the image (nxn) once through the FCNN and output is (mxm) prediction. This the architecture is splitting the input image in mxm grid and for each grid generation 2 bounding boxes and class probabilities for those bounding boxes. Note that bounding box is more likely to be larger than the grid itself. From paper:
We reframe object detection as a single regression problem, straight from image pixels to bounding box coordinates and class probabilities.
A single convolutional network simultaneously predicts multiple bounding boxes and class probabilities for those boxes. YOLO trains on full images and directly optimizes detection performance. This unified model has several benefits over traditional methods of object detection.
First, YOLO is extremely fast. Since we frame detection as a regression problem we don’t need a complex pipeline. We simply run our neural network on a new image at test time to predict detections. Our base network runs at 45 frames per second with no batch processing on a Titan X GPU and a fast version runs at more than 150 fps. This means we can process streaming video in real-time with less than 25 milliseconds of latency.
Second, YOLO reasons globally about the image when making predictions. Unlike sliding window and region proposal-based techniques, YOLO sees the entire image during training and test time so it implicitly encodes contextual information about classes as well as their appearance. Fast R-CNN, a top detection method, mistakes background patches in an image for objects

because it can’t see the larger context. YOLO makes less than half the number of background errors compared to Fast R-CNN. [14]
Third, YOLO learns generalizable representations of objects. When trained on natural images and tested on artwork, YOLO outperforms top detection methods like DPM and R-CNN by a wide margin. Since YOLO is highly generalizable it is less likely to break down when applied to new domains or unexpected inputs.
Our network uses features from the entire image to predict each bounding box. It also predicts all bounding boxes across all classes for an image simultaneously. This means our network reasons globally about the full image and all the objects in the image. The YOLO design enables end-to-end training and real-time speeds while maintaining high average precision.
Our system divides the input image into an $S \times S$ grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object.
Each grid cell predicts B bounding boxes and confidence scores for those boxes. These confidence scores reflect how confident the model is that the box contains an object and also how accurate it thinks the box is that it predicts. Formally we define confidence as Pr (Object) $*$ IOU . If no object exists in that cell, the confidence scores should be zero. Otherwise we want the confidence score to equal the intersection over union (IOU) between the predicted box and the ground truth.
Each bounding box consists of 5 predictions: x, y, w, h, and confidence. The (x, y) coordinates represent the center of the box relative to the bounds of the grid cell. The width and height are predicted relative to the whole image. Finally the confidence prediction represents the IOU between the predicted box and any ground truth box. Each grid cell also predicts C conditional class probabilities, Pr (Classi |Object). These probabilities are conditioned on the grid cell containing an object. We only predict one set of class probabilities per grid cell, regardless of the number of boxes B.
At test time we multiply the conditional class probabilities and the individual box confidence predictions, which gives us class-specific confidence scores for each box. These scores encode both the probability of that class appearing in the box and how well the predicted box fits the object

Changes to loss functions for better results is interesting. Two things stand out:

1. Differential weight for confidence predictions from boxes that contain object and boxes that don’t contain object during training.

2. Predict the square root of the bounding box width and height to penalize error in small object and large object differently.

Our network has 24 convolutional layers followed by 2 fully connected layers. Instead of the inception modules used by GoogLeNet, we simply use 1×1 reduction layers followed by 3×3 convolutional layers
Fast YOLO uses a neural network with fewer convolutional layers (9 instead of 24) and fewer filters in those layers. Other than the size of the network, all training and testing parameters are the same between YOLO and Fast YOLO.

We optimize for sum-squared error in the output of our model. We use sum-squared error because it is easy to optimize, however it does not perfectly align with our goal of maximizing average

precision. It weights localization error equally with classification error which may not be ideal. Also, in every image many grid cells do not contain any object. This pushes the “confidence” scores of those cells towards zero, often overpowering the gradient from cells that do contain objects. This can lead to model instability, causing training to diverge early on. To remedy this, we increase the loss from bounding box coordinate predictions and decrease the loss from confidence predictions for boxes that don’t contain objects. We use two parameters, λcoord and λnoobj to accomplish this. We set λcoord = 5 and λnoobj = .5.
Sum-squared error also equally weights errors in large boxes and small boxes. Our error metric should reflect that small deviations in large boxes matter less than in small boxes. To partially address this we predict the square root of the bounding box width and height instead of the width and height directly.[14]
YOLO predicts multiple bounding boxes per grid cell. At training time we only want one bounding box predictor to be responsible for each object. We assign one predictor to be “responsible” for predicting an object based on which prediction has the highest current IOU with the ground truth. This leads to specialization between the bounding box predictors. Each predictor gets better at predicting certain sizes, aspect ratios, or classes of object, improving overall recall.

	Pascal 2007 mAP	**Speed**	
DPM v5	33.7	.07 FPS	14 s/img
R-CNN	66.0	.05 FPS	20 s/img
Fast R-CNN	70.0	.5 FPS	2 s/img
Faster R-CNN	73.2	7 FPS	140 ms/img
YOLO	63.4	45 FPS	22 ms/img

First of all it puts an overlay grid on the image and tries to find that if there is any object within the box or not. If it can find object, it puts high confidence value to that and low confidence value if no object. Along with this the model classifies the selected objects parallel. It then takes the similarly classified boxes in a cluster and takes a bigger boundary. In this way it can classify object along detecting the boundary just by a single pass.

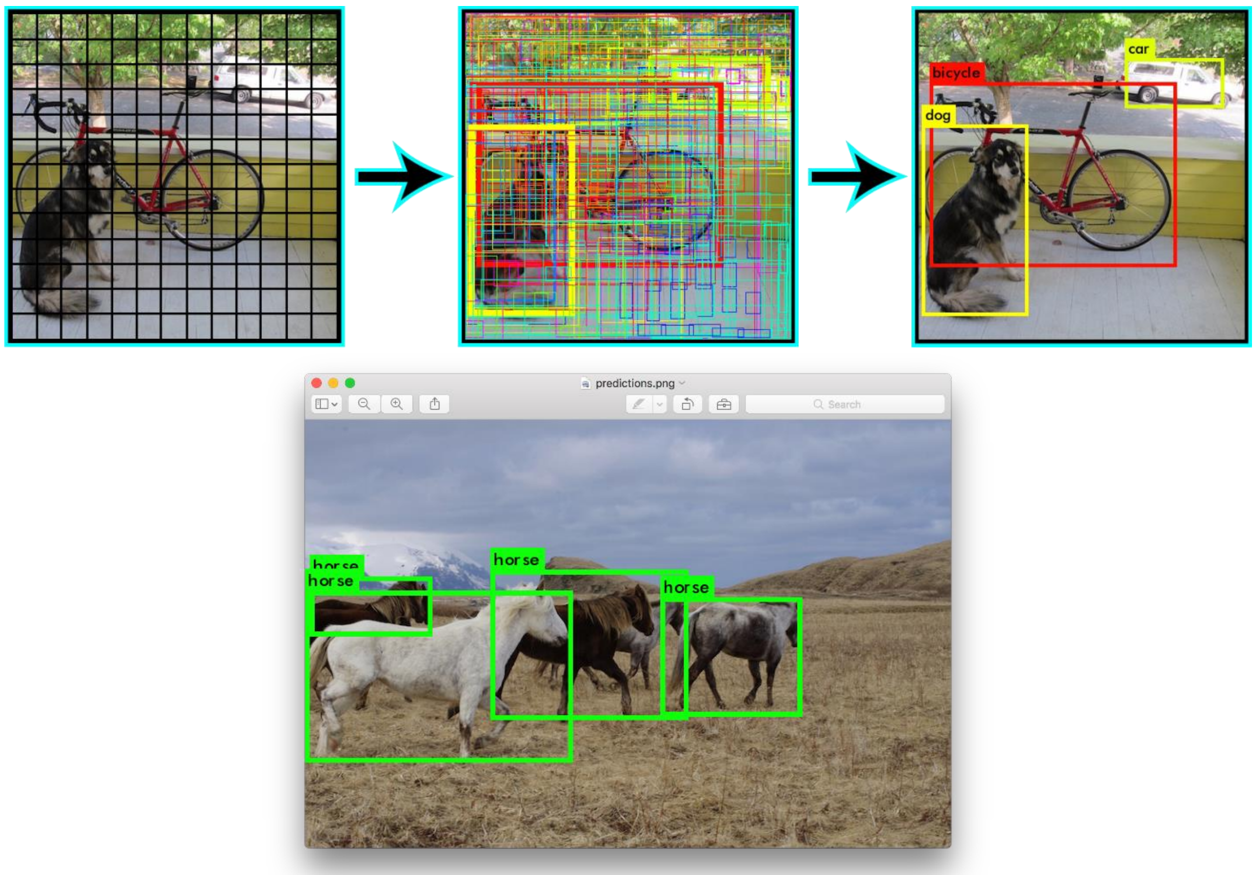


Figure 4.2: YOLO explanation [16]

By default, YOLO only displays objects detected with a confidence of .25 or higher. Putting the threshold to 0, we can see the defined object of all the possible boxes like the picture below.

4.2 Results:

4.2.1 TensorFlow

Two type of network models were used so that a comparative study could be possible. One was of the Convolutional Neural Network type and the other was a Simple 2-Layer model. The models were given shape through weight initialization, convolution and pooling, dropout and a classifier. Their compared accuracy is shown in the figure

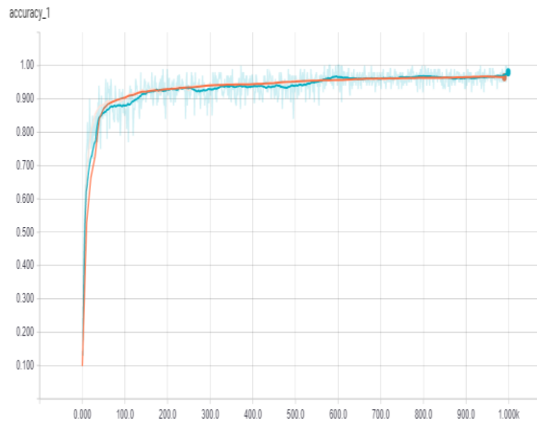


Figure 4.3: Accuracy CNN Model

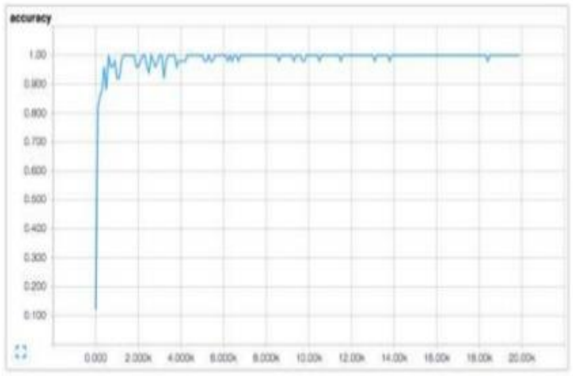


Figure 4.4: Accuracy Simple 2 Layer Mode

The goal of a loss function is to minimize the loss rate of a model as much as possible. We can easily understand the credibility of any particular model by looking at its loss function graph. A figure showing their loss functions is given below:

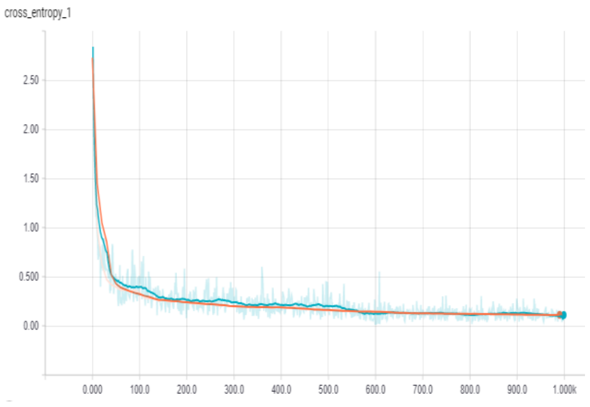


Figure 4.5: Loss Function Simple 2 layer model

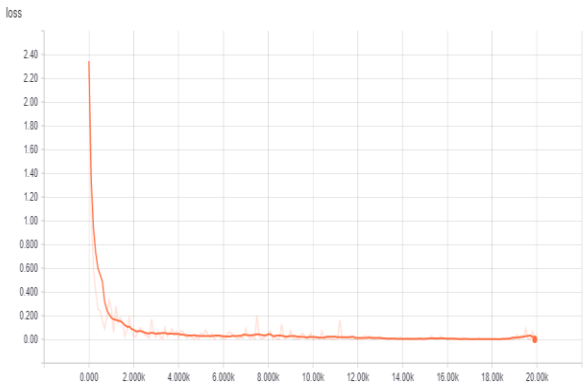


Figure 4.6: Loss Function CNN model

Again the comparisons were done by varying the parameters such as the number of convolutional network, the number of fully connected network and the learning rate. A figure to show the comparisons is given below:

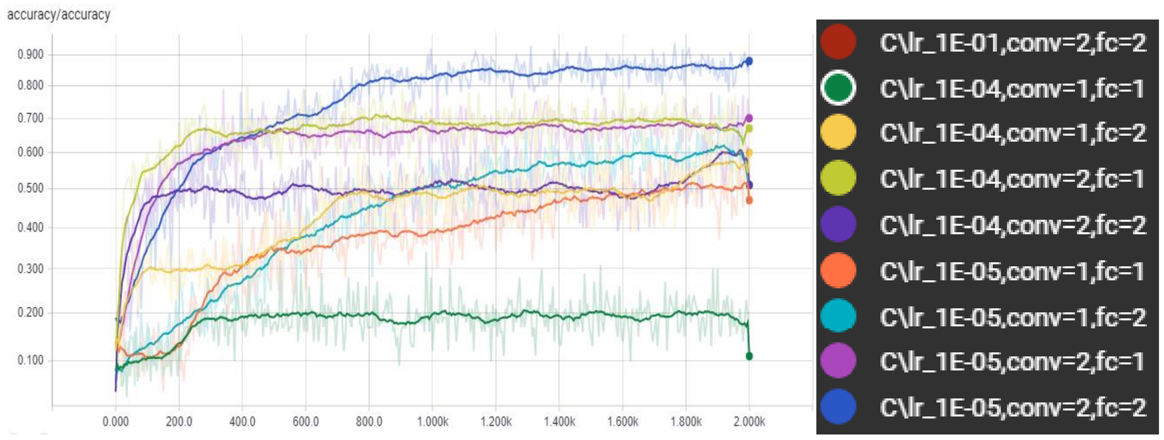


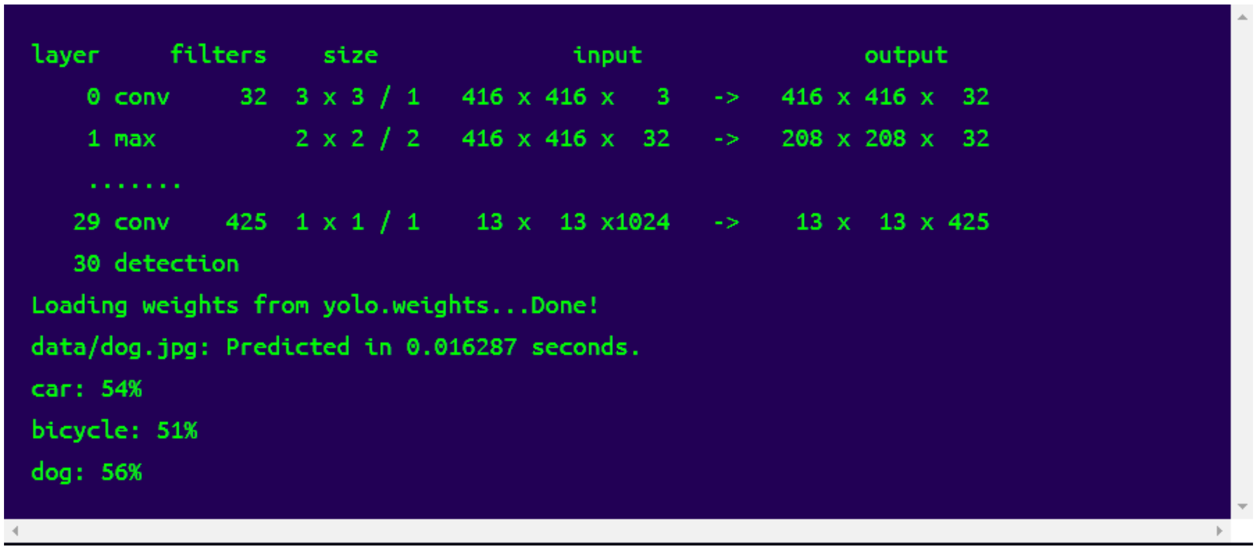
Figure 4.7: Accuracy Comparison by using different values of parameters

It is clear from the above figure that using 2 convolutional layers and two fully connected layers outputs a good result.

4.2.2 YOLO: DARKNET

DarkNet is an open source neural network framework written in C and CUDA. It is fast, easy to install and supports CPU and GPU computation.

Using this network we worked on object detection and classification together. To detect the object we used the pre trained weight values using Microsoft VOC 2007 dataset. For object classification, pretrained weight values of the model using ImageNet dataset were used. Using this framework the results could be generated very fast using very few commands and predictions were given using softmax approach.



```
layer     filters    size              input                output
    0 conv     32  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  32
    1 max          2 x 2 / 2   416 x 416 x  32   ->   208 x 208 x  32
    .......
   29 conv    425  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x 425
   30 detection
Loading weights from yolo.weights...Done!
data/dog.jpg: Predicted in 0.016287 seconds.
car: 54%
bicycle: 51%
dog: 56%
```

Figure 4.8: DarkNet performance on object detection & classification (1)

The objects were detected in the following manner:

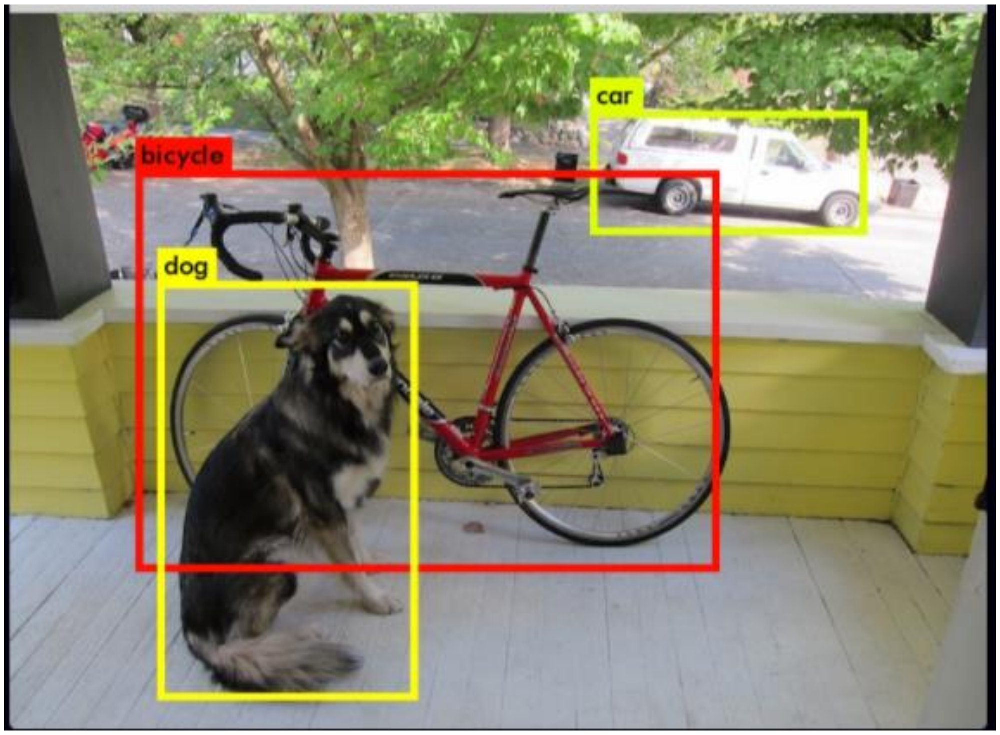


Figure 4.9:; Darknet performance on object detection & classification(2)

The model worked fine in detecting objects which were not collided. But if the object were collided it didn’t produce perfect detection of the object.

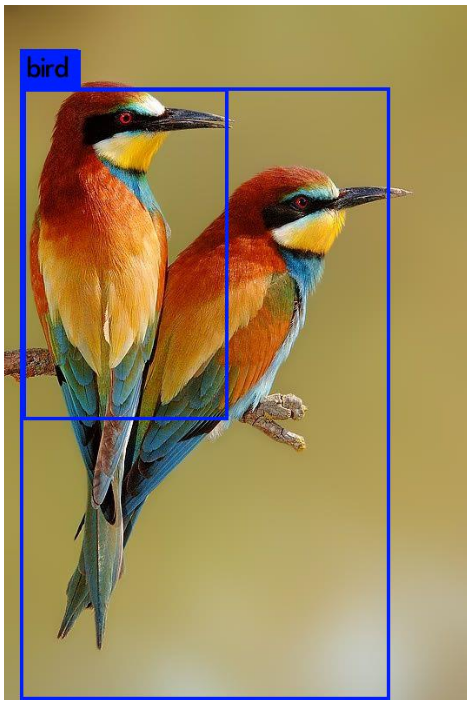


Figure 4.10: DarkNet performance on object detection & classification (3)

For example in the above image there are two birds in collided manner. The model can detect the bird in front, but it can't detect the second bird separately and detects two birds in a single box.

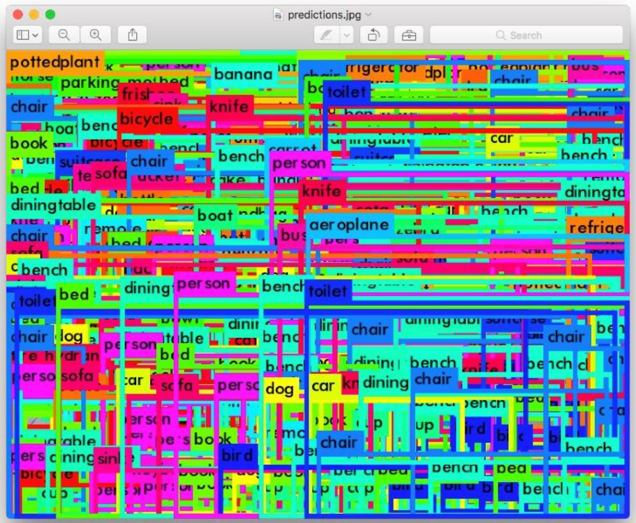


Figure 4.11: YOLO performance using zero threshold [16]

4.2.2.1 Biggest advantages:

- Speed (45 frames per second—better than real-time)
- Network understands generalized object representation (This allowed them to train the network on real world images and predictions on artwork was still fairly accurate).
- Faster version (with smaller architecture)—155 frames per sec but is less accurate.
- open source

4.2.2.2 Limitations of YOLO

YOLO imposes strong spatial constraints on bounding box predictions since each grid cell only predicts two boxes and can only have one class. This spatial constraint limits the number of nearby objects that our model can predict. The model struggles with small objects that appear in groups, such as flocks of birds. Since the model learns to predict bounding boxes from data, it struggles to generalize to objects in new or unusual aspect ratios or configurations. Again the model also uses relatively coarse features for predicting bounding boxes since the proposed architecture has multiple downsampling layers from the input image. Finally, while training on a loss function that approximates detection performance, the loss function treats errors the same in small bounding boxes versus large bounding boxes.

Chapter 5

5 Conclusion

5.1 Summary

Upon working with deep learning we can come to a conclusion that, it is a very promising branch of machine learning. It can be utilized to implement image processing systems that can output state of the art results. Although the internal architecture and work flow might seem a bit tedious and complex, using deep learning to create neural network models is gradually becoming the go to solution for modern image classification, objection classification and other problems present in the image processing domain. Throughout of thesis we came across various applications of deep learning ranging from recognizing handwriting to colorization of black and white images. Our thesis domain was limited to the problem of image classification using convolutional neural network. A convolutional neural network contains a series of layers performing relevant tasks of identifying features and give out a certain probability specifying an objects class. Such a network is considered to be a 'deep' network if there are more than 3 layers. In our thesis research we worked with such networks implementing them with the help of deep learning libraries: TensorFlow and Yolo. Our research rested mainly on experimenting with the parameters of such a network and compare the results.

5.2 Future Work

- Focusing on a particular image classification field and compare the results of previous approaches with Deep Learning methods.

- Use **transfer learning method** to use pre-trained models and propose possible improvements for specific image classification task

Bibliography

[1] *Colorful image colorization.* **Zhang, Richard, Phillip Isola, and Alexei A. Efros. 2016.** s.l. : European Conference on Computer Vision, 2016.

[2] *Deep neural networks in machine translation: An overview.* **Zhang, Jiajun, and Chengqing Zong. 2015.** s.l. : IEEE Intelligent Systems 30.5 : 16-25., 2015.

[3] *Deep visual-semantic alignments for generating image descriptions.* **Karpathy, Andrej, and Li Fei-Fei. 2015.** s.l. : IEEE Conference on Computer Vision and Pattern Recognition, 2015.

[4] *Dropout: A Simple Way to Prevent Neural Networks from Overfitting.* **Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov. 2014.** s.l. : The Journal of Machine Learning Research, 15(1), 1929-1958., 2014.

[5] *Explain images with multimodal recurrent neural networks.* **Mao, J., Xu, W., Yang, Y., Wang, J., & Yuille, A. L. 2014.** s.l. : arXiv preprint arXiv:1410.1090., 2014.

[6] *Generating sequences with recurrent neural networks.* **Graves, Alex. 2013.** s.l. : arXiv preprint arXiv:1308.0850, 2013.

[7] *Generative adversarial nets.* **Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. 2014.** s.l. : Advances in neural information processing systems (pp. 2672-2680), 2014.

[8] *ImageNet Classification with Deep Convolutional Neural Networks.* **Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton. 2012.** s.l. : Advances in neural information processing systems (pp. 1097-1105)., 2012.

[9] **Nielsen, Michael.** *Neural Networks and Deep Learning.*

[10] *Rectified Linear Units Improve Restricted Boltzmann Machines.* **Vinod Nair, Geoffrey E. Hinton. 2014.** s.l. : Proceedings of the 27th international conference on machine learning (ICML-10) (pp. 807-814)., 2014.

[11] *Scalable object detection using deep neural networks.* **Erhan, Dumitru, Christian Szegedy, Alexander Toshev, and Dragomir Anguelov. 2014.** s.l. : IEEE Conference on Computer Vision and Pattern Recognition, pp. 2147-2154, 2014.

[12] *Visualizing and Understanding Convolutional Neural Networks.* **Matthew D. Zieler, Rob Fergus. 2014.** s.l. : European conference on computer vision (pp. 818-833). Springer International Publishing., 2014.

[13] *Visually indicated sounds.* **Owens, A., Isola, P., McDermott, J., Torralba, A., Adelson, E.H. and Freeman, W.T. 2016.** s.l. : IEEE Conference on Computer Vision and Pattern Recognition (pp. 2405-2413), 2016.

[14] *YOLO9000: Better. Faster, Stronger.* **Redmon, J., & Farhadi, A. 2016.** s.l. : arXiv preprint, 2016.

[15] machinelearningmastery.com. '8 Inspirational Applications of Deep Learning', July 14, 2016. [Online]. Available: https://machinelearningmastery.com/inspirational-applications-deep-learning/. [Accessed: 02-April-2017]

[16] Pjreddie.com. 'YOLO: Real-Time Object Detection'. [Online]. Available: https://pjreddie.com/darknet/yolo. [Accessed: 03-Nov-2017]

[17] Towardsdatascience.com. 'YOLO — you only look once, real time object detection explained'. 2016. [Online]. Available: https://towardsdatascience.com/yolo-you-only-look-once-real-time-object-detection-explained-492dc9230006. [Accessed: 06-Nov=2017]

[18] Hackernoon.com. 'Challenges in Deep Learning'. 2017. [Online]. Available: https://hackernoon.com/challenges-in-deep-learning-57bbf6e73bb. [Accessed: 15-Mar-2017]

[19] Adeshpande3.github.io. 'A Beginner's Guide to Understanding Convolutional Neural Networks'. 2016. [Online}. Available: https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks. [Accessed: 05-Mar-2017]

[20] Towardsdatascience.com. 'A Survey of Deep Learning Frameworks'. 2017. [Online] Available: https://towardsdatascience.com/a-survey-of-deep-learning-frameworks-43b88b11af34. [Accessed: 05-Nov-2017]

[21] Tensorflow.org. 'Getting started with TensorFlow'. 2017. [Online] Available: https://www.tensorflow.org/get_started/get_started. [Accessed: April-2017]