AUTOMATIC RECOGNITION OF CAPTCHAS USING NEURAL NETWORKS

A Thesis

presented to

the Faculty of Natural and Applied Sciences

at Notre Dame University-Louaize

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

MARYBEL GEAGEA

MAY 2019

Notre Dame University - Louaize

Faculty of Natural and Applied Sciences

Department of Computer Science

We hereby approve the thesis of

Marybel Geagea

Candidate for the degree of Master of Science in Computer Science

[illegible]	[Signature]
Dr. Hikmat Farhat	Supervisor, Chair
[illegible]	[Signature]
Dr. Khaldoun khaldi	Committee Member
[illegible]	[Signature]
Dr. Khalil Challita	Committee Member

Declaration

This is to ensure that I, Marybel Geagea, student at Notre Dame University, studying a Master of Computer Science, have submitted this thesis in part fulfillment of the requirements for my Master's degree.

Furthermore, I hereby certify that this entire material belonging to my MCS dissertation, which I am now submitting for assessment, is entirely my own work and has not been taken from the substance of others spare to the degree that such work has been referenced to recognize inside the content of my own development and elaboration.

Finally, I announce that this thesis has not been submitted in part or in whole to any other University for assessment.

Acknowledgments

I would like to thank the Notre Dame University, for the graduate assistantship opportunity and for affording me the un-imaginable chance to complete my study here. I will not forget to thank my thesis advisor Dr. Hikmat Farhat whom for without his patience, guidance, support and understanding, I would never have made it this far.

Table of Contents

List of Figures

List of Tables

Abstract

The Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA) is a test used to differentiate between humans and machines. In this thesis, we have implemented a deep neural network architecture to automatically recognize CAPTCHAs. The main idea behind this thesis is to test the reliability of test-based CAPTCHAs in differentiating between humans and machines. We utilized convolutional neural networks instead of the conventional strategies of CAPTCHA breaking which, typically, use segmentation techniques. Our model is comprised of 8 layers to learn text-based images; five of which are used for convolution and the remaining three are dense feedforward layers. We attempted various parameters to our model, including dropout rate, maxpooling value and the number of samples used. We generated a CAPTCHA dataset of two different types; colored and gray scaled images CAPTCHAs and had the capacity to get accuracy levels of 98.6% and 100%, respectively.

Chapter 1: Introduction and Problem Definition

1.1 Introduction to the General Problem

In many situations, especially when creating or logging into Internet accounts, it is important to differentiate between humans and machines. For example, bots were sometime used to automatically create hundreds of thousands of email accounts and use them to send SPAM email. Bots can flood the servers of any company that allows free registration service. If not properly managed such bots can automatically perform thousands of registrations at the same time thus creating a denial of service attack. Also, all password protected services are prone to dictionary attacks where bots can try thousands of passwords from a dictionary. All the above problems can be mitigated to a large degree if there is a mechanism to differentiate between humans and bots. Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA) is one successful mechanism that has been used to solve the problems listed above. Using CAPTCHAs no client action is completed unless it can prove that it is human and not a bot. To be useful CAPTCHAs are designed to be easy to recognize by humans but hard for bots.

1.2 Problem Definition

With the recent advances in deep learning for pattern recognition the question arises if CAPTACHAs cannot be subverted by machine learning method and thus render them useless. Specifically, if bots can use machine learning techniques to automatically recognize CAPTCHAs then the such test will fail in differentiating between humans and bots.

1.3 Research Objectives

The goal of this thesis is to determine the efficacy of text-based CAPTCHAs, which are basically images of distorted text. We found that text based CAPTCHAs do not provide a good test for differentiating between humans and machine since a machine learning algorithm can automatically recognize word based CAPTCHAs. More specifically, we were able, using neural networks, to automatically determine word based CAPTCHAs with a large degree of accuracy.

1.4 Approach and Main Results

We have used deep convolution networks to automatically identify text-based CAPTCHAs.

Our work shows that text-based CAPTCHAs are no longer a reliable method to differentiate between humans and machines.

1.5 Thesis Organization

In the second chapter some of the CAPTCHA's types are presented and a detailed definition of the CAPTCHA is given with a description of the algorithm, previous work and the motivation behind this thesis. In chapter 3 a detailed understanding on all the machine learning concepts used to make this project successful are explained in which we used neural networks. In chapter 4 results are presented for different experiments and for different parameters showing where we obtained the highest accuracies and the best results and some of the CAPTCHAs that we were not able to break or the machine algorithm gave wrong results with comparisons and charts showing all the results. Finally, we conclude with chapter 6.

Chapter 2: Background and Motivation

2.1 Types of CAPTCHAs

CAPTCHA also known as the Completely Automated Public Turing test to tell Computers and Humans Apart is a challenge test that will make sure that a human is trying to access this website and not a bot or another computer (Gafni & Nagar, 2016).

It is formed of a sequence of distorted characters some are only letters and other are a combination of letters and numbers as one image.

Although the term was conceived in 2003 by Louis von Ahm, Manuel Blum, Nicholas J. Hopper, and John Langford (Von Ahn, Blum, Hopper, & Langford, 2003), it was first developed in 1997 by two parallel working groups.

It was first used to get rid of computer bots that started in the early 90’s spamming the internet.

Hackers and spammers exist trying to steal information or do some illegal work by utilizing their computer bots and they can be in control of many PCs and devices specially using the internet, so we are all disposed to many threats and vulnerabilities in our daily life.

Computer bots are programs used and written for a special purpose, one example in YAHOO a program was written and was able to create as many accounts as it wants on its own and in a fast time, so CAPTCHAs were created for this reason, to be able to identify human from computer.

CAPTCHAs come in different shapes and types and sizes, used against spams and differ in the difficulty to solve and some are harder to solve than others.

Most text CAPTCHAs deployed in our real life applications nowadays are broken with the use of advanced segmentation and OCR techniques. CAPTCHAs can be more secure by the addition of noise and distortion and the arrangement of the characters to be tighter. But these measures used on these characters will also make it harder to be recognized by humans and this will lead to a higher error percentage among people (Sharma & Seth, 2015).

We have a limit on how much distortion and how much noise is available that human will tolerate in any CAPTCHA challenge.

Two main issues must be considered while designing a scheme; security and usability (Soumya & Abraham, 2014). CAPTCHAs must be as easy as possible on humans to be able to identify them but at the same time it must be as hard as possible on bots so that it can’t be broken because we are using CAPTCHAs in the first place so that bots can’t access websites easily.

The design used should be robust against any attack that is automated and used by a computer program or a bot. The tradeoff between these two main issues is difficult and must be balanced, and the high rate of successfully solving a CAPTCHA challenge by humans must be kept as high as possible while the possibility that a program can solve it correctly should be as low as possible (Chow & Susilo, 2017).

Cybercrimes are growing fast and most of these crimes come from an automated computer or a bot threating and attempting to reach unauthorized webpages. According to Symantec for example spams received daily on our emails form 75.8% of the total number of emails sent. Not all spams are computer based but 82.2% of these spams are generated by bots and this is the main reason why websites tend to use CAPTCHAs so that bots are restricted from entering these sites. More than 3.5 million webpages globally use CAPTCHAs for a security reason especially in fill-in forms, comment writing and tickets buying types of websites and more than 300 million tests are submitted by users daily (Gafni & Nagar, 2016).

Many popular websites are vulnerable and subject to attacks by computers or programs also known as bots and bellow many examples of these attacks are elaborated to understand exactly what bots are capable of doing and why CAPTCHAs are that popular (Sharma & Seth, 2015; Yalamanchili & Rao, 2011).

1. Blogs comment Spam

   Bogus comments submitted by programs that bloggers are familiar with exist. Usually these comment's aim is to increase the rank of some websites in search engines. Many of these comments are ads that users may have the curiosity to open and explore and they are known as comment spam. Sign up procedures are not important anymore and any user can enter a comment without signing in but only after submitting and passing a CAPTCHA test.

2. Website registrations

   Free email services are available and popular everywhere, and many big companies like YAHOO! and Microsoft offer this kind of service. After being that popular these companies suffered from bot's attack on their service and bots where able to create thousands of emails each minute mainly used to send spams to users. These services found that the only solution for these threats is by using CAPTCHAs before being able to sign up to any account making the email generation limited only to legitimate users

3. Online polls

   Many competitions where results are based on human voting using internet services is available nowadays. These websites used an IP address restriction so that users can only vote once and won't be able to repeat it using the same IP address or device. Many programs are written for voting purposes and allow bots to vote repeatedly hence results are fake. These online polls cannot be trusted anymore unless they use CAPTCHAs to make sure that only humans are voting.

4. Dictionary attacks

   Passwords are used everywhere, protecting our own privacy and restricting others from entering unauthorized systems or applications. Used in home networks to login to our router and have access to the internet, used in emails and many other services. A program is written, and a dictionary is available having a big number of combinations that will try each at a time until it matches the used password. But recently we noticed that even if I am a legitimate and I enter by mistake a wrong password several times the webpage will have to ask me to solve a CAPTCHA to proceed and be able to try again entering the correct password. And this technique used will also block dictionary attacks after many unsuccessful attempts and the program won’t be able to continue its job without solving the CAPTCHA test.

5. Denial of service attack

   Another known attack used by bots is sending requests for a purpose to exhaust or even worse it can shut down a web server preventing legal users to access needed information. These examples were a motivation so that these websites used CAPTCHAs and tried as much as possible to prevent their server to be subject to a denial of service attacks.

Plenty of CAPTCHA examples exist but the most common ones used nowadays are: word CAPTCHAs, audio, 3D, branded and math solutions (Soumya & Abraham, 2014).

In this report we will be focusing on word CAPTCHAs that also come in many types, difficulty level to solve and size.

Standard CAPTCHA or the most common one used is a set of distorted scrambled word which the user must identify each element of and enter it in a special slot, sometimes this word is too hard and not recognized even by human, so he has the option to ask for a new CAPTCHA word that he can identify.

Over the past years, numerous CAPTCHAs have emerged because it has been found that many of these schemes were vulnerable and broken and that’s why experts are always trying to design a more robust scheme with a variety of types.

Five types of CAPTCHAs can be classified (Singh & Pal, 2014):

1. Based on puzzle
2. Based on image
3. Based on audio
4. Based on video
5. Based on text

1. BASED ON PUZZLE

In puzzle based CAPTCHAs, the user must solve a riddle consisting of many chunks of images and he should combine these chunks to form a normal picture, identify a missing part of a given image or a given puzzle that includes two displaced pieces that the user should fix.

This time of CAPTCHAs is not dependent on any language and can be considered easy for humans to solve but difficult for bots.

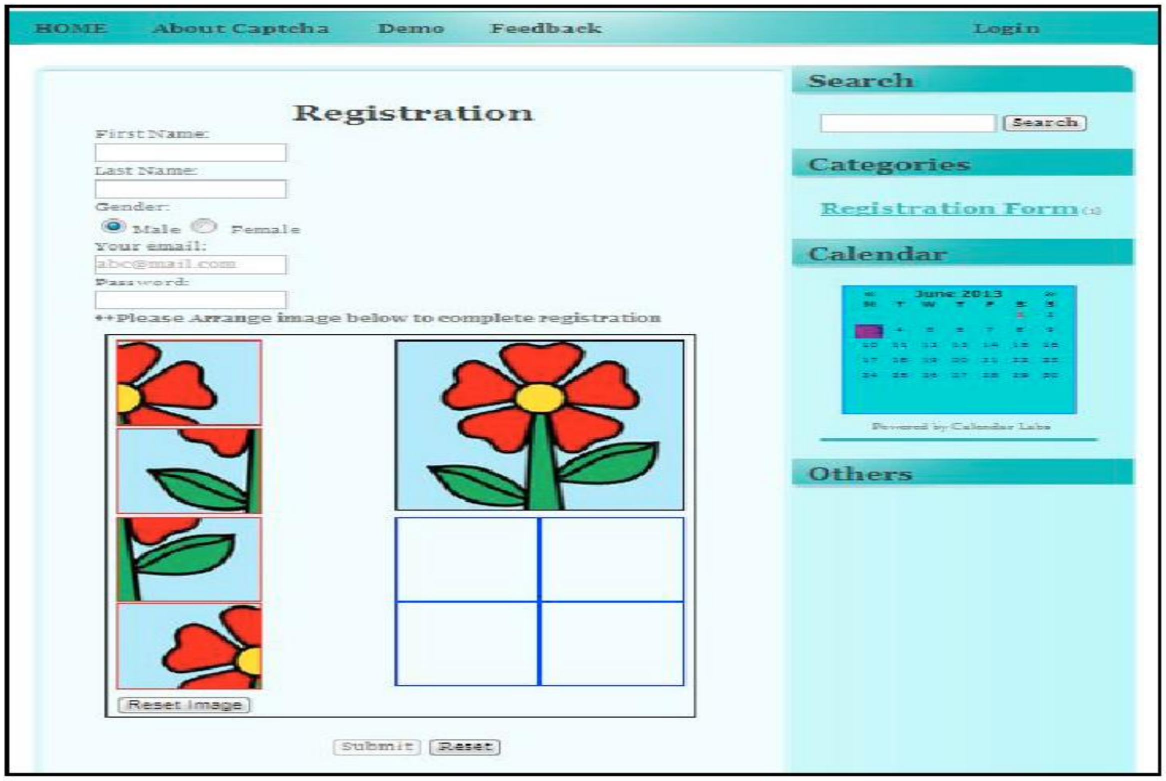
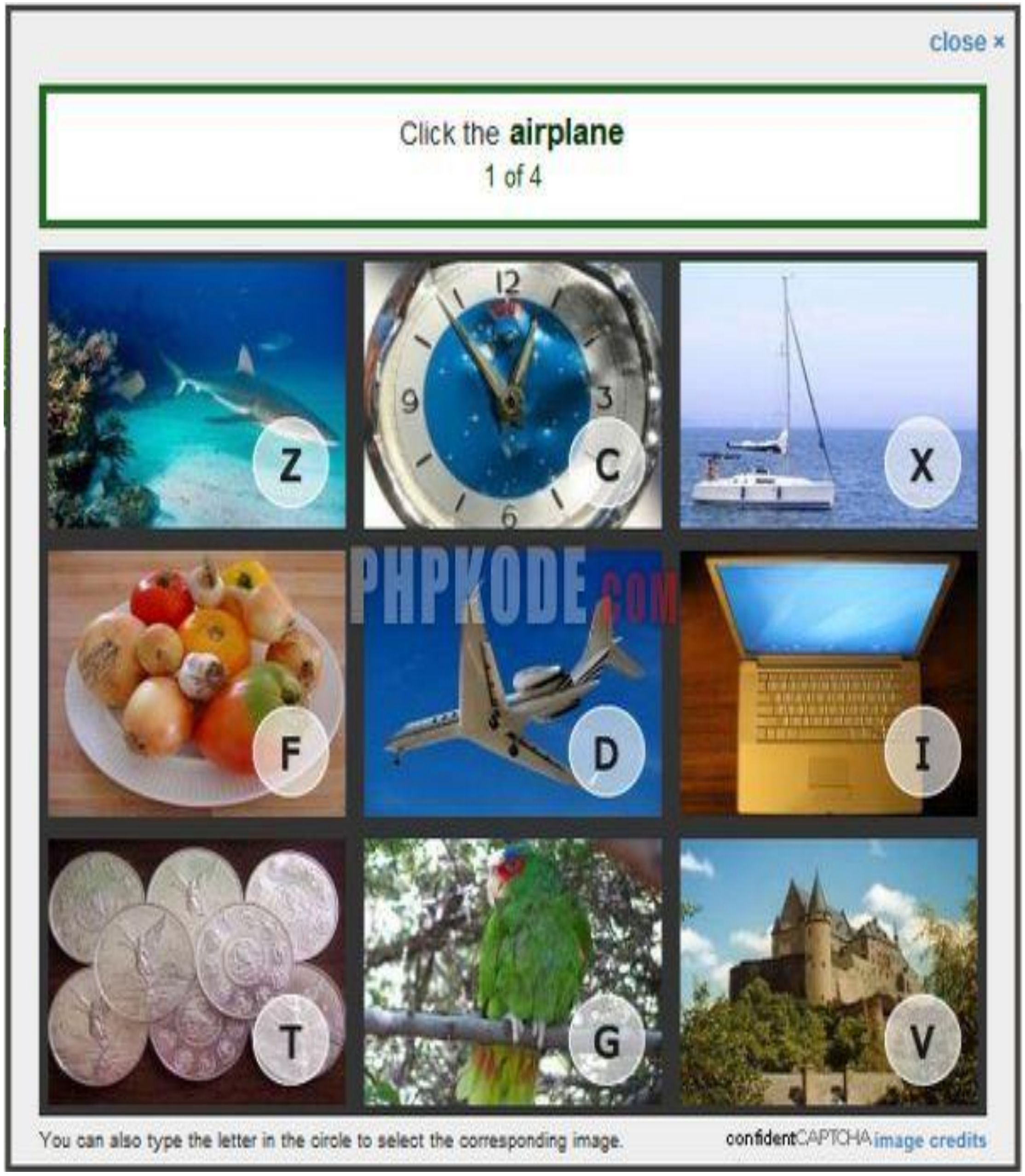


Figure 2.1 Based on puzzle CAPTCHA

2. BASED ON IMAGE

In image CAPTCHAs, user must identify a particular image from similar images sometimes combined with words or identify to which concept or object a group of images belong

This type of CAPTCHA as well is considered easy on humans but not that easy on bots to solve



You can also type the letter in the circle to select the corresponding image.

Figure 2.2 Based on image CAPTCHA

3. BASED ON AUDIO

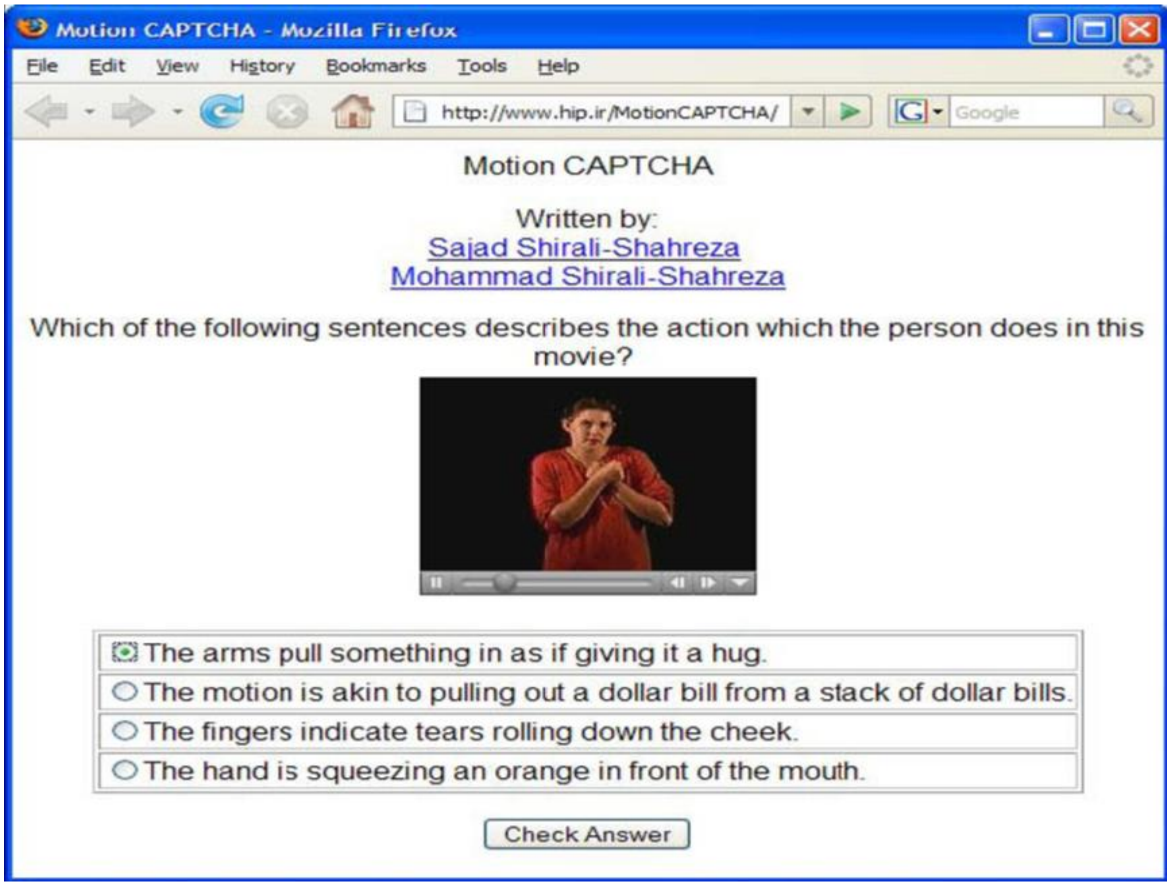
In audio CAPTCHAs, a voice that introduce a specific word is presented to the user and he should be able to recognize what he heard and type it to solve it.

An important advantage in this type of CAPTCHAs is that it can be used by users having visual impairment.

4. BASED ON VIDEO

Rarely used type of CAPTCHAs in which user is shown a short clip of a person that is performing an action and some sentences describing many actions are presented to the user from which he should recognize the correct description that corresponds to the video.

Although it can be considered an easy test for humans, but it requires the user to have a good knowledge in the English language.


Motion CAPTCHA - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.hip.ir/MotionCAPTCHA/
Google

Motion CAPTCHA

Written by:
Sajad Shirali-Shahreza
Mohammad Shirali-Shahreza

Which of the following sentences describes the action which the person does in this movie?

[✓ The arms pull something in as if giving it a hug.]
The motion is akin to pulling out a dollar bill from a stack of dollar bills.
The fingers indicate tears rolling down the cheek.
The hand is squeezing an orange in front of the mouth.

Figure 2.3 Based on video CAPTCHA

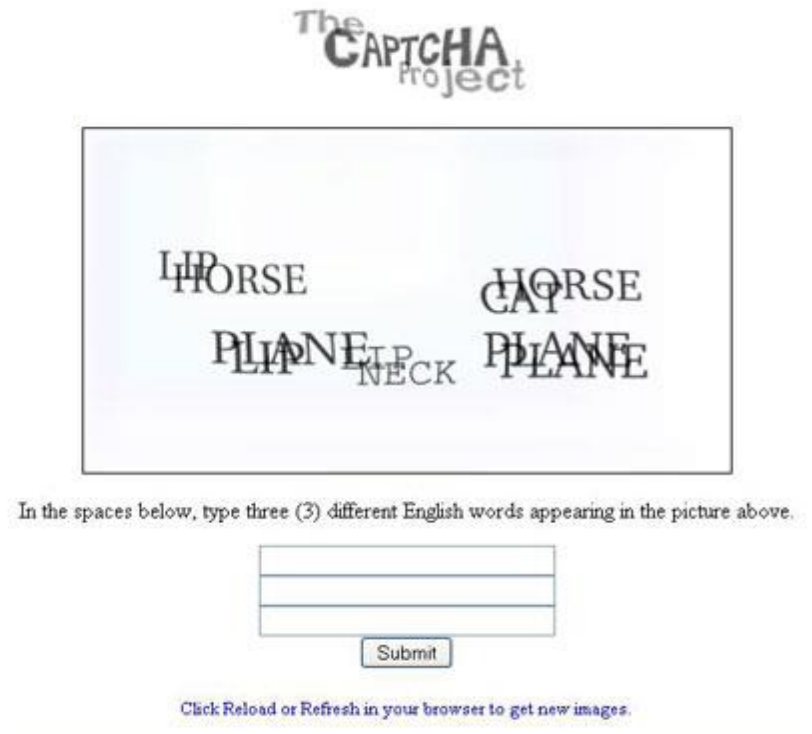
5. BASED ON TEXT

Discussed previously in this report, simple to implement in which we have a sequence of letters or a combination of letters and digits presented to the user where some modifications are applied to the characters such as scattering, rotation and noise but still readable to the user but not easy for bots to guess. Text based CAPTCHAs has a big number of methods where each has been improved with time.

Bellow some of these methods will be presented (Hasan, 2016)

a. Gimpy CAPTCHA

Just as described above it's a sequence of characters presented in a distorted image, in this method the technique used is to add black and white lines with a non-linear adjustment and then as usual ask the user to guess the word and type it in a special slot in the website. Later this method was enhanced in association with Yahoo and was mostly used in chat rooms preventing spammers from posting ads and to prevent bots from creating automated free e-mail accounts.



In the spaces below, type three (3) different English words appearing in the picture above.

Click Reload or Refresh in your browser to get new images.

Figure 2.4 Example of gimpy CAPTCHA

b. Scatter type Method

In this method instead of adding noise and modifications to the whole sequence, it is segmented into characters that can be letters or numbers. And after segmentation each of these pieces is modified aside by adding noise and rotation and other techniques. It was used because the previous methods were broken and bots where able to deduce the input with a high precision rate. But in this method it wasn’t that easy at the beginning since the letters are hard to separate knowing that each letter is segmented into many small pieces.

Additionally, this method chooses letters randomly and they whole word does not make any sense and cannot be found in any dictionary.

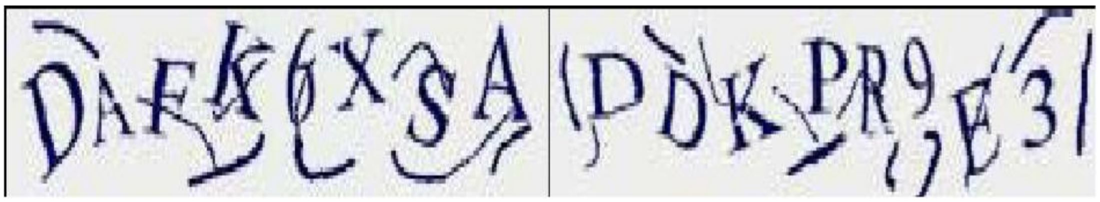


Figure 2.5 Example of scatter type CAPTCHA

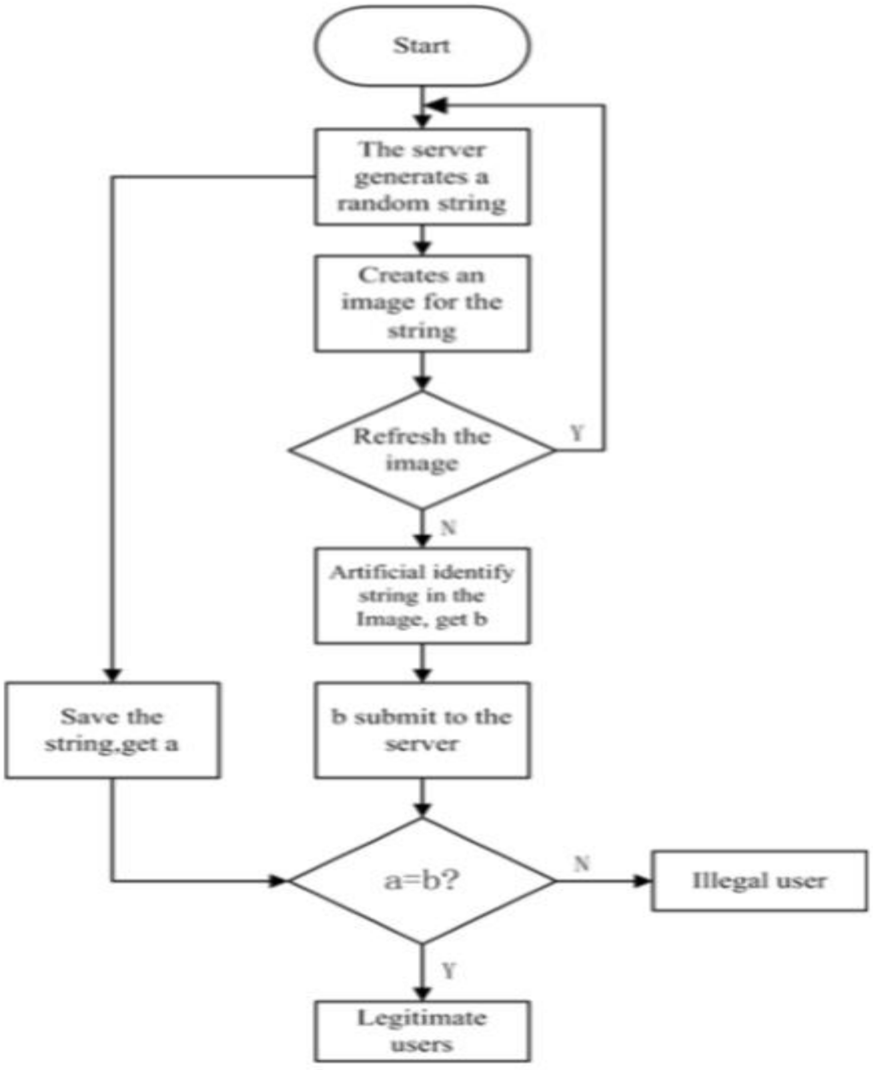

Start
The server generates a random string
Creates an image for the string
Refresh the image
Y
N
Artificial identify string in the Image, get b
Save the string,get a
b submit to the server
a=b?
N
Illegal user
Y
Legitimate users

Figure 2.6 CAPTCHA's basic principle

Using the schematic diagram above used and explained by (Lei, 2015), the working process of the mostly used type of the CAPTCHAs and the one discussed mainly in this report which is the text based CAPTCHA can be understood.

And this process works as follows:

Whenever a user enters the website and to proceed with the basic function he needs a CAPTCHA, a random function is generated by the server and a random string is produced. This string will be saved on the server and then some noise and interference is added to this string in an image form.

This image is embedded and posted in the web page so that the user can see, try to recognize the initial string and enter his result in a special slot made for this purpose and in this way it’s not easy on bots to decode this image and understand the string easily.

Once this image is posted, user has 2 choices either recognize it correctly and proceed or ask for another combination because this one is not easy and too much noise exist in this image that this user finds not readable.

If the user asks for another string then same steps will be repeated overwriting the previous string, if not the user enters the recognized string and this server will have to compare it to the initial one saved before transforming this string to an image.

If this entered string by the user matches to saved one, the server will presume that it is a legal user trying to enter the webpage and will let him proceed with his work. But if that’s not the case the server will think that it is an illegal user or bot trying to access the webpage and won’t let him do that.

2.2 Descriptions of the Used Algorithms

We have used deep neural networks to implement an automatic recognition of CAPTCHAs software. In particular, we have used TensorFlow framework to build a deep convolution neural network. We have also used CAPTCHA generator.

2.3 Previous Work in the Subject

CAPTCHA’s that are text based like the ones we are using in this project are broadly studied by scholars.
A method used by many is segmenting the text and extracting the characters that are recognized as an example Haichang Gao etc and for this method CAPTCHA achieved a 78% success rate as an overall individual character rate (Wang, Yang, Zhu, & Liu, 2018).
Zhao Wang and Zitong Cheng used a denoising and segmentation technique for CAPTCHA’s containing colored interference lines, rotation distortion and adhesion and got a 78.5% accuracy rate (Wang et al., 2018).

Mori and Malik created an algorithm to break EZ-Gimpy CAPTCHA’s having a success rate of 92% and a 33% success rate in breaking the Gimpy CAPTCHAs. They used hand-tunes systems and sophisticated object recognition algorithms (Garg & Pollett, 2016).
Wang Yang et al. used k-nearest neighbor or what is known by KNN technique to recognize verification codes (Hu, Chen, & Cheng, 2018).

2.4 Research Motivation

Our main goal in thesis was to test the resiliency of test-based CAPTCHAs against start-of-the-art deep learning methods. Specifically, we wanted to give a definite answer about the suitability of using text-based CAPTCHAs with the huge advance in pattern recognition using deep learning.

Chapter 3: Neural Networks

Before starting this chapter I would like to note that all the information found in this part are learned from two books (Bishop, 1995; Goodfellow, Bengio, & Courville, 2016)

Artificial Intelligence, in the early days, solved many problems that people can find tough like chess or graph problems. And it was a challenge solving easy problems like identifying objects. It is basically learning or teaching this machine from experience and let it try to find and recognize patterns for new problems. At first it learns from knowledge found in the world in formal languages and model it to be able to use inference rules and deduce new problems.

One example used to make machines and software learn is by using neural network which is the approach used in this thesis.

Many variants and techniques are used in neural networks like feedforward, recurrent or convolution networks, etc…

These models are similar more or less and they all use the same building block which is the perceptron or neuron.

So neural networks is a graph or network of perceptrons or neurons connected together.

A perceptron can be compared to a human neuron and it is a non-linear computational unit.

It has n inputs $x_1,\ldots, x_n$ which each has its linked weight w_i.

First step done is computing the quantity $\sum w_i x_i$. It then adds a bias b and we obtain $a= \sum w_i x_i +b$.

Finally a non linear function f to this result $y=f(a)$.

$a= \sum wixi +b$ can be written in a vector form as $a=w^T.x+b$ where the “ . ” is the dot product of the two vectors; the transpose of w vector and x.

Let’s consider the normal plane to vector w, as shown in the picture below u is parallel to w having a length $\frac{b}{|w|}$

Any vector x can be written as x= u+v, and we can have w.x= w.u + w.v

But having the two vectors w and u parallel, their dot product is a constant and we can have w.u = b

Therefore w.x = b + w.v

For the points on the dotted plane w.v =0 because these vectors are perpendicular

On the left w.v is negative and to the right w.v is positive having the same direction.

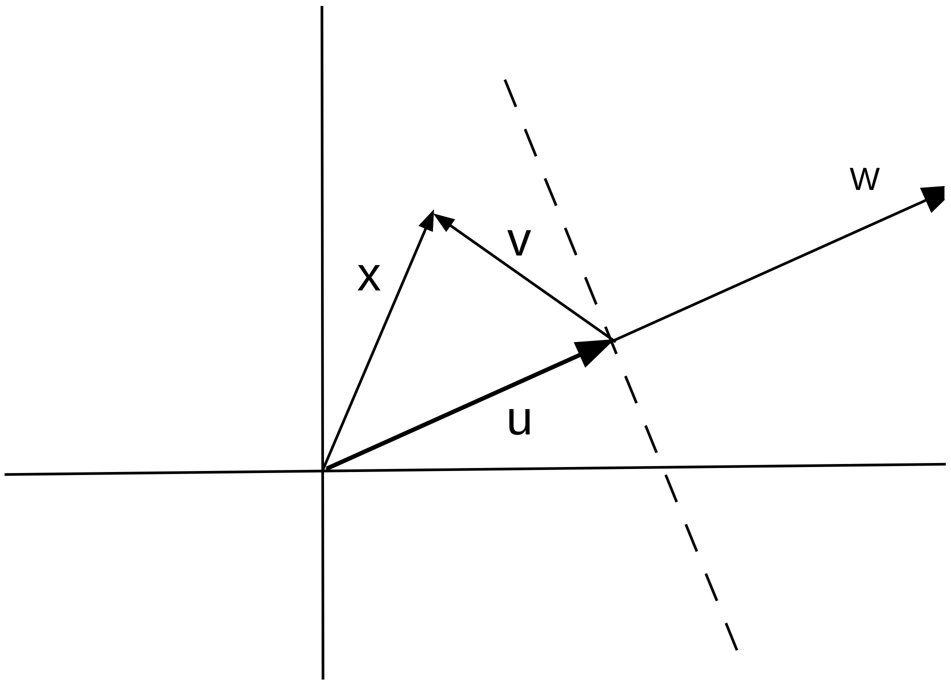


Figure 3.1 Geometric interpretation of the perceptron

Our aim is to find an algorithm that will correctly classify any point if it belongs to class C1 and C2

Now let’s consider we have two sets H1⊆ C1 and H2⊆ C2 properly labeled, the algorithm used learns from these labeled data and obtains optimal values for b and the weights w used. After this learning stage this algorithm with its fixed values can be used to predict any new data that is not labeled and should be able to classify this new input in its corresponding set to which it belongs to.

The inputs will be extended from $x_0,\dots,x_n$ to $x_0,\dots,x_n,x_{n+1}$ and the same for w to $w_0,\dots,w_n,w_{n+1}$ where $X_{n+1}=1$ and $w_{n+1}=-b$

And in this case our equation will become $w^T.x = 0$

As a simple example the algorithm works as follows:

1. If x belongs to H1 and $w.x \leq 0 \rightarrow$ do nothing
2. If x belongs to H2 and $w.x \geq 0 \rightarrow$ do nothing
3. If x belongs to H1 and $w.x > 0 \rightarrow w=w-x$
4. If x belongs to H2 and $w.x < 0 \rightarrow w=w+x$
5. All points in H1 will be converted $x= -x$
6. We only have 2 rules: 2 and 4.

Python is a dynamically typed language in which variables are bound to objects at execution time, ideal for prototyping because of its interpreted nature and open source.

The perceptron algorithm will at first be used to learn the logical AND function which has 4 possible inputs.

```
4
 0 0 0
 0 1 0
 1 0 0
 1 1 1
```

In this example "4" denotes the number of lines which is four, as for the lines, first 2 denote our inputs and the third denotes the output of the AND function.

Obviously, we have 2 classes in this example; the first one which has an output 0 and the second has an output 1. First three sets belong to the first class and the last set belongs to the second class.

The number of samples is very small, so we can iterate repeatedly until this algorithm converges and doesn't change anymore.

Using the extended method discussed above, we will solve for $w_1x_1+w_2x_2+w_3x_3=0$. And since $x_3=1$, the line equation of the above equation is as follows: $x_2= -\frac{w1}{w2}x_1 - \frac{w3}{w2}$.

And the plot of the above example is shown in the below picture.

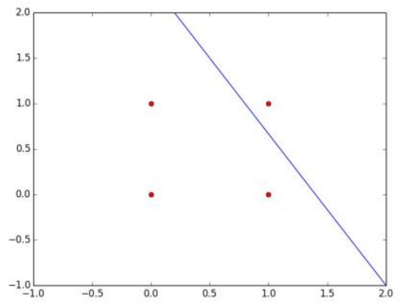


Figure 3.2 Separable class example

Another example that identifies ships will be explained in this section. A modified version of what was explained earlier will be used to be able to identify images that contains ships in it.

The dataset used is called CIFAR-10 available online and it contains images of 10 different types or classes objects which one of these types is for ships.

Our perceptron used will try to learn to identify ships and the model once if finishes learning will hopefully output 1 if the image is for a ship and 0 if not.

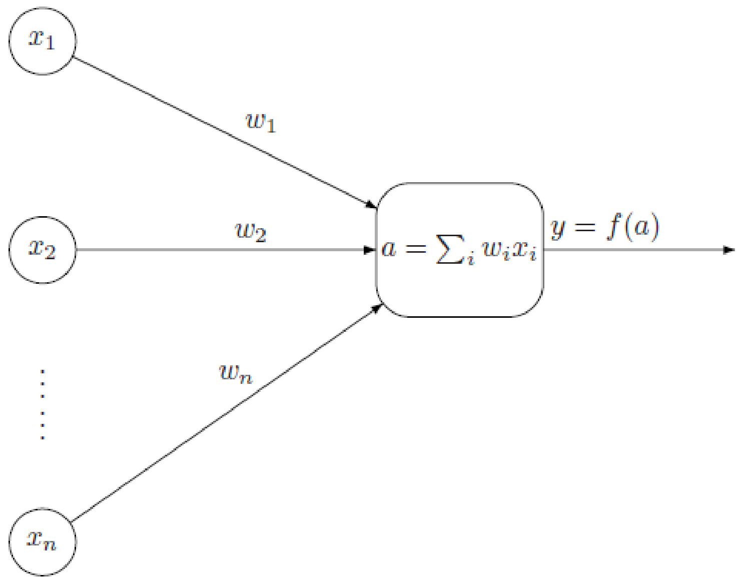


Figure 3.3 Perceptron non-linear representation

First the output used in this model is called the sigmoid function which introduces non-linearity, it is differentiable and that's an important property as we will see later.

$$\sigma(z) = \frac{1}{1 + e^{(-z)}}$$

To represent images in our model we will use a three-dimensional array; the first two represents the pixels of the image, as for the third dimension it gives the Red-Green-Blue value of each pixel.

In the case of the dataset used in this example CIFAR-10, images are represented by 32x32 pixels and this image should be flattened.

The first 1024 values are the Red values of my pixels, the next 1024 represents the Green values and finally the last 1024 are the Blue values.

So, the input used in this perceptron will be a vector of 3072 dimension with values between 0 and 255.

The dataset used contains 60000 images and as said we have 10 classes in this dataset with 6000 images in each class.

After converting my images to a vector, the second step done is dividing the dataset into training images for the learning process and testing images for the testing process to make sure that our algorithm converged and is working fine. In this example 50000 images are taken for training and 10000 for testing.

The training set is also divided into 5 batches or files, when the data is read from a given file we are reading 10000 samples of vectors and these samples will be processed at once as a single batch rather than iterating 10000 times.

Let x^i be the input vector for sample i with size n=3072.

Then m samples or vectors are represented as a matrix.

$$X = \begin{bmatrix} x_1^1 & x_1^2 & \dots & x_1^m \\ x_2^1 & x_2^2 & \dots & x_2^m \\ \dots & \dots & \dots & \dots \\ x_n^1 & x_n^2 & \dots & x_n^m \end{bmatrix}$$

The output of our model becomes

$\hat{Y} = \sigma\,[w^T.X + b]$

Where σ is the sigmoid function introduced earlier.

It helps sometimes to visualize the above computation

$$[\hat{y}_1 \dots \hat{y}_m] = \sigma\left([w_1 \dots w_n] \cdot \begin{bmatrix} x_1^1 & x_1^2 & \dots & x_1^m \\ x_2^1 & x_2^2 & \dots & x_2^m \\ \dots & \dots & \dots & \dots \\ x_n^1 & x_n^2 & \dots & x_n^m \end{bmatrix} + b\right)$$

So according to the equation above $\hat{y}_i$ is the output for the ith sample.

As for the σ function applied to the matrix as shown above, it is this same matrix where each element is obtained by applying this σ function to it.

Also this value b added to a matrix will be added separately to each element in this matrix.

And now we can compute the value ŷ given input x and if we can determine the weights w and our bias b.

Since our output is binary, 1 if the image is a ship and 0 otherwise then ŷ is interpreted as follows; the probability that y = 1 given x.

In other words, given an image represented by x, what is the probability that this image is a ship?

If ŷ= 0.99, the image is most probably of a ship and if ŷ=0.2, the image is unlikely one of a ship.

Now to choose the optimal values for the weights and bias, my error should be minimized. This error of our prediction ŷ for a given sample x depends on how closely it predicts the value of y also known as the label associated with each sample.

Cross-entropy, a function used in this example to calculate the difference is given by

$-(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$

And the average error over all the samples is as follows:

$$E = -\frac{1}{m}\sum_{i=1}^{m} y^i \log \hat{y}^i + (1 - y^i)\log(1 - \hat{y}^i) = \frac{1}{m}\sum_{i=1}^{m} E^i$$

As said before to find the optimal solution for w and b in our problem, the error or average error E should be minimized. This technique is called the gradient descent and we illustrate this idea in a 2-dimentional curve in a picture below to be able to understand it.

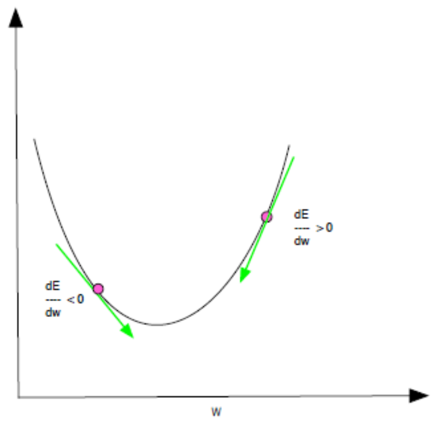


Figure 3.4 2-d Gradient Descent representation

From the figure illustrated above we can obviously see that if the derivative of my error over w is positive, w should be decreased to get to my minimum value, and if this derivative is negative w should be increased.

In both cased my w is updated as follows w= w – α$\frac{dE}{dw}$

Where α determines the rate used of which w is updated.

And clearly when $\frac{dE}{dw} = 0$ then w will be unchanged, and we get to the minimum value that we want to reach.

To compute this gradient descent equation, we need $\frac{\partial E}{\partial w}$ and $\frac{\partial E}{\partial b}$.

Since E = $\sum_{i=1} E^i$, it is enough to get the derivatives of each value of i E^i and then average this value over the number of samples.

Using the chain rule we can write $\frac{\partial E^i}{\partial w} = \frac{\partial E^i}{\partial \hat{y}^i}\frac{\partial \hat{y}^i}{\partial w}$

We compute each term

$$\frac{\partial E^i}{\partial \hat{y}^i} = \frac{\partial}{\partial \hat{y}^i}\left[-y^i \log \hat{y}^i - (1-y^i)\log(1-\hat{y}^i)\right]$$
$$= -\frac{y^i}{\hat{y}^i} + \frac{1-y^i}{1-\hat{y}^i}$$

On the other hand $z^i = w^T . x^i + b$ and $\hat{y}^i = \sigma (z^i)$ then

$$\frac{\partial \hat{y}^i}{\partial w} = \frac{\partial \hat{y}^i}{\partial z^i}\frac{\partial z^i}{\partial w}$$

We now have

$$\begin{aligned}\frac{\partial \hat{y}^i}{\partial z^i} &= \frac{\partial}{\partial z^i}\frac{1}{1+e^{-z^i}} = \frac{e^{-z^i}}{(1+e^{-z^i})^2}\\ &= \frac{-1+(1+e^{-z^i})}{(1+e^{-z^i})^2} = \hat{y}^i - \hat{y^i}^2\end{aligned}$$

And since $\partial z^i/\partial w = x^i$ then

$$\begin{aligned}\frac{\partial \hat{y}^i}{\partial w} &= (\hat{y^i} - \hat{y^i}^2) \cdot x^i\\ &= \hat{y}^i(1-\hat{y}^i) \cdot x^i\end{aligned}$$

Combining all the partial results that we got

$$\begin{aligned}\frac{\partial E^i}{\partial w} &= \left[-\frac{y^i}{\hat{y}^i} + \frac{1-y^i}{1-\hat{y}^i}\right]\left[\hat{y}^i(1-\hat{y}^i)\cdot x^i\right]\\ &= \left[-y^i(1-\hat{y}^i) + (1-y^i)\hat{y}^i\right]\cdot x^i\\ &= (\hat{y}^i - y^i)\cdot x^i\end{aligned}$$

To get the derivatives of E we average over all samples

$$\frac{\partial E}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^i - y^i)x^i$$

We can reuse all the partial computations and the fact that $\partial z^i/\partial b = 1$ to get

$$\frac{\partial E}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^i - y^i)$$

Typically when we apply the algorithm we would have m samples. Let x_j^i be the jth sample. Similarly, y^i is the label of the ith sample.

Recall that we average over all samples

$$\frac{\partial E}{\partial w_j} = \sum_{i=1}^{m}(\hat{y}^i - y^i)x_j^i$$

In vector notation

$$dw = \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \dots \\ \frac{\partial E}{\partial w_n} \end{bmatrix} = \frac{1}{m} \begin{bmatrix} x_1^1 & \dots & x_1^m \\ x_2^1 & \dots & x_2^m \\ \dots & \dots & \dots \\ x_n^1 & \dots & x_n^m \end{bmatrix} \cdot \begin{bmatrix} \hat{y}^1 - y^1 \\ \hat{y}^2 - y^2 \\ \dots \\ \hat{y}^m - y^m \end{bmatrix}$$

Or using the transpose of $[(\hat{y}^m - y^1)\dots(\hat{y}^m - y^m)]$

$$dw = X.\ (\hat{y} - y)^T$$

Similarly

$$\frac{\partial E}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^i - y^i)$$

And now to summarize all the above derivations and equations; ŷ ,y and w are now vectors, dw is a column vector then the formulas we need are

$$\begin{aligned} \hat{y} &= \sigma(w \cdot x + b) \\ dw &= \frac{1}{m}x \cdot (\hat{y} - y)^T \\ db &= \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^i - y^i) \\ b &= b - \alpha db \\ w &= w - \alpha dw^T \end{aligned}$$

Where α as mentioned earlier is the learning rate.

Once our iterations finish, our computation produces optimal values for the weight w and bias b and these values will be used later in the testing process to predict an output and compute an accuracy value for my algorithm.

This approach used with a single perceptron was found working fine with sets that are separable and in fact it has a poor performance on 2-Dimensional data sets. So, it can work with simple separable sets and can get an accuracy up to 99% on the learning set and approximately 83% on the testing set. But with more advanced and harder problems another approach is needed because this one is trying to separate data by using a plane.

Multilayer perceptron known as MLP is a group of perceptrons arranged in layers. The output of the first perceptron which was before our dataset and solution will now be considered as an input to the next perceptron which will also feed the perceptron coming just after it until we don’t have any perceptrons left and this is our output. When a loop is present in the graph and an output is connected to an input node, this network is known as recurrent network, and when no loops are available our network is called feedforward.

Bellow is a figure showing an example of a feedforward network containing two hidden layers which can be variable depending on the application or problem we have

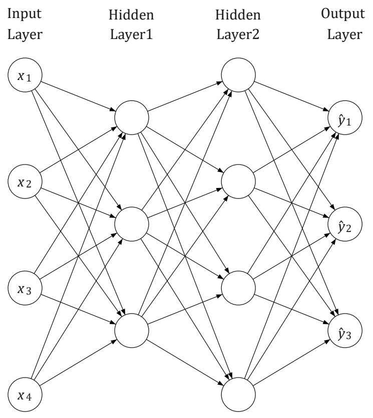


Figure 3.5 Feedforward network with two hidden layers

As shown in the picture we can have $x_1,\ldots,x_m$ inputs and $\hat{y}_1,\ldots,\hat{y}_k$ outputs.

The last layer called L has $a_i^L = \hat{y}_i$. As discussed earlier each hidden layer will have an output a_i^l that will be used as an input for the next layer having an equation similar to the one used in a simple one layer perceptron $a_i^l = \sigma(\sum_i w\,a\ + b)$ where w and b are the weights and bias of the layer l and a is the output of layer l-1, as for σ it is a nonlinear function which is bounded and nondecreasing.

Each node can be treated as a single layer perceptron discussed earlier regardless of which layer it is in.

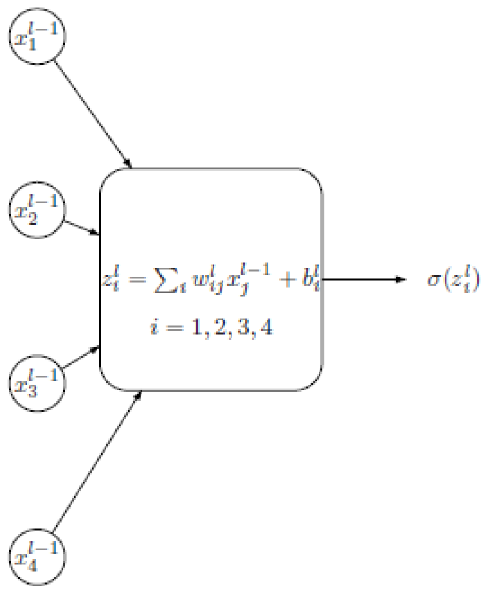


Figure 3.6 Computation at a node

An example of a single hidden layer is given in this report which is also called a shallow network.

By convention the first layer is not counted, we have one hidden layer and one final layer, so it is considered as a two-layer network.

In this example our network has two inputs, the hidden layer consists of four nodes and the final layer of one node. The subscript of each node denotes its output, the superscript denotes the layer; a_2^1 is the output of the second node in layer 1.

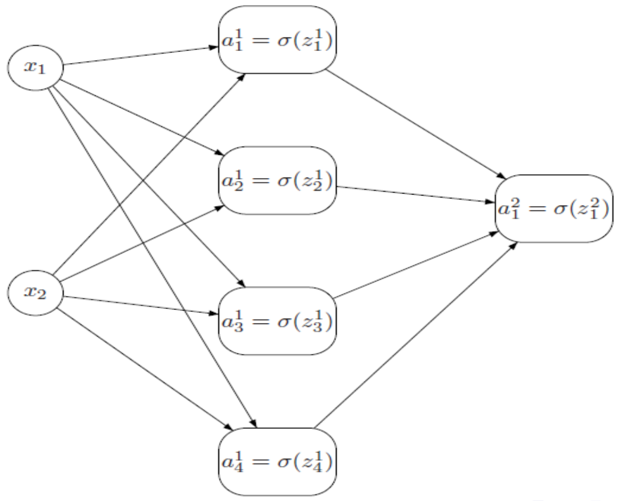


Figure 3.7 Single hidden layer shallow network

The node computations in layer one can be written as:

$$z_1^1 = w_{11}^1 * x_1 + w_{12}^1 * x_2$$
$$z_2^1 = w_{21}^1 * x_1 + w_{22}^1 * x_2$$
$$z_3^1 = w_{31}^1 * x_1 + w_{32}^1 * x_2$$
$$z_4^1 = w_{41}^1 * x_1 + w_{42}^1 * x_2$$

And the output of the first layer in vector form is written as:

$$\begin{bmatrix} a_1^1 \\ a_2^1 \\ a_3^1 \\ a_4^1 \end{bmatrix} = \begin{bmatrix} \sigma(z_1^1) \\ \sigma(z_2^1) \\ \sigma(z_3^1) \\ \sigma(z_4^1) \end{bmatrix}$$

Similarly, for the second layer where my inputs now are the outputs of the first layer and now we have a final output which has the following equation:

$$\hat{y} = a_1^2 = \begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 & w_{14}^2 \end{bmatrix} \cdot \begin{bmatrix} a_1^1 \\ a_2^1 \\ a_3^1 \\ a_4^1 \end{bmatrix}$$

We will write the computational equations to include m samples. All my sample dependent quantities such as a, z, x have a single superscript which is the layer number as mentioned before.

As for the subscript a_{jk}^i j is the node label and k is the number of samples.

Using the above notation, we can include all the samples in a single vector operation.

Using the subscript s for sample s we have. For every 1≤ s ≤ m

$$\begin{bmatrix} z_{1s}^1 \\ z_{2s}^1 \\ z_{3s}^1 \\ z_{4s}^1 \end{bmatrix} = \begin{bmatrix} w_{11}^1 & w_{12}^1 \\ w_{21}^1 & w_{22}^1 \\ w_{31}^1 & w_{32}^1 \\ w_{41}^1 & w_{42}^1 \end{bmatrix} \cdot \begin{bmatrix} x_{1s} \\ x_{2s} \end{bmatrix}$$

We can column-stack al the examples together to get:

$$\begin{bmatrix} z_{11}^1 & \cdots & z_{1m}^1 \\ z_{21}^1 & \cdots & z_{2m}^1 \\ z_{31}^1 & \cdots & z_{3m}^1 \\ z_{41}^1 & \cdots & z_{4m}^1 \end{bmatrix} = \begin{bmatrix} w_{11}^1 & w_{12}^1 \\ w_{21}^1 & w_{22}^1 \\ w_{31}^1 & w_{32}^1 \\ w_{41}^1 & w_{42}^1 \end{bmatrix} \cdot \begin{bmatrix} x_{11} & \cdots & x_{1m} \\ x_{21} & \cdots & x_{2m} \end{bmatrix}$$

In vector form and having A^0=X, we can write

$$\begin{aligned} Z^1 &= W^1 \cdot A^0 + B^1 \\ A^1 &= \sigma(Z^1) \\ Z^2 &= W^2 \cdot A^1 + B^2 \\ A^2 &= \sigma(Z^2) \end{aligned}$$

The reason why we choose to use a nonlinear function in the algorithm is that when we tried to use a linear one, no matter how many layers there are in the network, it would be equivalent to the perceptron model.

From the previous example if we set $A^i = f(Z^i) = Z^i$ we get:

$$\begin{aligned} Z^1 &= W^1 \cdot X + B^1 \\ A^1 &= f(Z^1) = Z^1 \\ Z^2 &= W^2 \cdot A^1 + B^2 \\ A^2 &= f(Z^2) = Z^2 \end{aligned}$$

And now replacing A^1 by its value we get:

$\hat{Y} = A^2 = Z^2 = W^2 . (W^1 .X + B^1) + B^2$

$= (W^2 . W^2). X + (W^2 . B^1 + B^2)$

$= W. X + B$

To do the gradient descent for the shallow network we need to compute the derivatives of the cost function with respect to the parameters. The loss is given by the cross-entropy function bellow:

$$\mathcal{L} = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

To do the optimization we use gradient descent in which the parameters are updated in the opposite direction of the derivative of the error function explained earlier in this report.

$$w = w - \alpha \frac{\partial E}{\partial w}$$
$$b = b - \alpha \frac{\partial E}{\partial b}$$

We will compute the derivatives with respect to the loss then average over all samples.

The derivative with respect to b^2 is as follows:

$$\frac{\partial \mathcal{L}}{\partial b^2} = \frac{\partial \mathcal{L}}{\partial a^2} \frac{\partial a^2}{\partial b^2}$$

We already know that $a^2 = \hat{y}$, so:

$$\frac{\partial \mathcal{L}}{\partial a^2} = \frac{-y}{a^2} + \frac{1-y}{1-a^2}$$

And since:

$$\begin{aligned} \frac{\partial a^2}{\partial b^2} &= \frac{\partial a^2}{\partial z^2} \frac{\partial z^2}{\partial b^2} \\ &= a^2(1-a^2) \end{aligned}$$

Thus:

$$\frac{\partial \mathcal{L}}{\partial b^2} = (a^2 - y)$$

As for the average over m samples becomes:

$$db^2 = \frac{1}{m} \sum_{s=1}^{m} (a_s^2 - y_s)$$

As for the derivative with respect to w^2:

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_i^2} &= \frac{\partial \mathcal{L}}{\partial a^2}\frac{\partial a^2}{\partial w_i^2} \\
&= \frac{\partial \mathcal{L}}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial w_i^2} = (a^2 - y)\frac{\partial z^2}{\partial w_i^2} \\
&= (a^2 - y)a_i^1
\end{aligned}
$$

Averaging over all samples we get:

$$
dw_i^2 = \frac{1}{m}\sum_{s=1}^{m}(a_s^2 - y_s)a_{is}^1
$$

$$
dw^2 = \frac{1}{m}(A^2 - Y) \cdot A^{1\,T}
$$

Derivative with respect to w^1:

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{w_{ij}^1} &= \frac{\partial \mathcal{L}}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial w_{ij}^1} \\
&= (a^2 - y)\frac{\partial z^2}{\partial w_{ij}^1}
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial z^2}{\partial w_{ij}^1} &= \frac{\partial}{\partial w_{ij}^1}\left[\sum_k w_k^2 a_k^1 + b^2\right] \\
&= \sum_k w_k^2 \frac{\partial \sigma(z_k^1)}{\partial w_{ij}^1} \\
&= \sum_k w_k^2 \sigma'_k \frac{\partial z_k^1}{\partial w_{ij}^1}
\end{aligned}
$$

But:

$$
z_k^1 = \sum_p w_{kp}^1 x_p + b_k^1
$$

$$
\frac{\partial z_k^1}{\partial w_{ij}^1} = \delta_{ki} x_j
$$

Replacing in the expression for $\frac{\partial z^2}{\partial w^1_{ij}}$ we get:

$$\frac{\partial \mathcal{L}}{\partial w^1_{ij}} = (a^2 - y) w^2_i \sigma'_i x_j$$

Averaging over m samples we get:

$$dw^1_{ij} = \frac{1}{m} \sum_s (a^2_s - y_s) w^2_i \sigma'_{is} x_{js}$$
$$dw^1 = \frac{1}{m} \left[\left({w^2}^T \cdot (A^2 - Y) \right) * \sigma' \right] \cdot X^T$$

And finally, the derivative with respect to b^1:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial b^1_i} &= \frac{\partial \mathcal{L}}{\partial a^2} \frac{\partial a^2}{\partial b^1_i} = \frac{\partial \mathcal{L}}{\partial a^2} \frac{\partial a^2}{\partial z^2} \frac{\partial z^2}{\partial b^1_i} \\ &= (a^2 - y) \frac{\partial}{\partial b^1_i} \left[\sum_k w^2_k a^1_k + b^2 \right] \\ &= (a^2 - y) \sum_k w^2_k \frac{\partial a^1_k}{\partial b^1_i} = (a^2 - y) \sum_k w^2_k \sigma'_k \frac{\partial z^1_k}{\partial b^1_i} \end{aligned}$$

$$\frac{\partial \mathcal{L}}{\partial b^1_i} = (a^2 - y) \sum_k w^2_k \sigma'_k \delta_{ki} \qquad = (a^2 - y) w^2_i \sigma'_i$$

Averaging over m samples we get:

$$db^1 = \frac{1}{m} \sum_s (a^2_s - y_s) {w^2}^T \sigma'$$

Let n_h be the number of hidden nodes and m be the number of samples:

$$\begin{aligned} Z^1 &= W^1 \cdot X + B^1 && (n_h, 2) \times (2, m) + (n_h, 1) = (n_h, m) \\ A^1 &= \sigma(Z^1) \\ Z^2 &= W^2 \cdot A^1 + B^1 && (1, n_h) \times (n_h, m) + (1, 1) = (1, n_h) \end{aligned}$$

$$db^2 = \frac{1}{m}\sum_s (a_s^2 - y_s) \qquad (1,1)$$

$$dw2 = \frac{1}{m}(A^2 - Y)\cdot {A^1}^T \qquad (1,m)\times(m,n_h) = (1,n_h)$$

$$db^1 = \frac{1}{m}\sum_s \left[{w2}^T \cdot (A^2 - Y)\right] * \sigma' \qquad \sum_s (n_h,1)\times(1,m) = (n_h,1)$$

$$dw^2 = \frac{1}{m}\left[\left({w2}^T \cdot (A^2 - Y)\right) * \sigma'\right]\cdot X^T \quad (n_h,1)\times(1,m)\times(m,2) = (n_h,2)$$

Now to compute the gradient we need to compute:

$$\frac{\partial E}{\partial w_{ij}^l}$$

And

$$\frac{\partial E}{\partial b_i^l}$$

For all w_{ij}^l and b_i^l

Since b_i^l appears explicitly only in z_i^l we can write:

$$\frac{\partial E}{\partial b_i^l} = \frac{\partial E}{\partial z_i^l}\frac{\partial z_i^l}{\partial b_i^l}$$

Recall that

$$z_i^l = \sum\nolimits_p w_{ip} a_p^{l-1} + b_i^l$$

Thus

$$\frac{\partial z_i^l}{\partial b_i^l} = 1$$

And it follows that:

$$\frac{\partial E}{\partial b_i^l} = \frac{\partial E}{\partial z_i^l} = \delta_i^l$$

We have defined that δ_i^l which we will refer to often.

Now computing the derivative with respect to w_{ij}^l

For the weights we have:

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}^l} &= \frac{\partial E}{\partial z_i^l}\frac{\partial z_i^l}{\partial w_{ij}^l}\\ &= \delta_i^l \frac{\partial z_i^l}{\partial w_{ij}^l}\end{aligned}$$

Since

$$z_i^l = \textstyle\sum_p w_{ip} a_p^{l-1} + b_i^l$$

Then:

$$\frac{\partial z_i^l}{\partial w_{ij}^l} = a_j^{l-1}$$

Therefore

$$\frac{\partial E}{\partial w_{ij}^l} = \delta_i^l a_j^{l-1}$$

Now to computing δ_i^l we note that z_i^l is used as an input through a_i^l to all nodes in level l+1 so:

$$\begin{aligned}\delta_i^l = \frac{\partial E}{\partial z_i^l} &= \sum_k \frac{\partial E}{\partial z^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_i^l}\\ &= \sum_k \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_i^l}\\ &= \sum_k \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial a_i^l}\frac{\partial a_i^l}{\partial z_i^l}\\ &= \sum_k \delta_k^{l+1} w_{ki}^{l+1}\sigma'(z_i^l)\end{aligned}$$

Where σ' is the derivative of the sigmoid function and we have used the fact that

$$z_k^{l+1} = \sum_p w_{kp}^{l+1} a_p^l + b_k^l$$

Summarizing the results so far:

$$\frac{\partial E}{\partial b_i^l} = \delta_i^l$$
$$\frac{\partial E}{\partial w_{ij}^l} = \delta_i^l a_j^{l-1}$$
$$\delta_i^l = \frac{\partial E}{\partial z_i^l} = \sum_k \delta_k^{l+1} w_{ki}^{l+1} \sigma'(z_i^l)$$

As we can see the computation of the derivatives of E depends on the computation of δ_i^l

The computation of δ_i^l for layer *l* can be computed if we know its value δ_k^{l+1} in the next layer. Therefore, the computation is called backpropagation.

Computing δ_i^l for the last layer l=L we can compute all other values. To do so we need an explicit expression of E

$$E = -\sum_k \left(y_k \log a_k^L + (1 - y_k) \log(1 - a_k^L) \right)$$

Then:

$$\delta_i^L = \frac{\partial E}{\partial z_i^L} = \frac{\partial E}{\partial a_i^L} \frac{\partial a_i^L}{\partial z_i^L} = \left[-\frac{y_i}{a_i^L} + \frac{1 - y_i}{1 - a_i^L} \right] \sigma'(z_i^L)$$

And since σ'(z_i^L) = a_i^L(1- a_i^L) we get δ_i^L=(a_i^L-y_i)

For a gaussian

$$E = -\frac{1}{2}\sum_k (y_k - a_k^L)^2$$

Then δ_i^L=(a_i^L-y_i) a_i^L(1- a_i^L)

To summarize the results in a vector form and since δ_i^l and a_i^l are column vectors the expressions:

$$\delta_i^l = \sum_k \delta_k^{l+1} w_{ki}^{l+1} \sigma'(z_i^l)$$
$$b_i^l = \delta_i^l$$
$$dw_{ij}^l = \delta_i^l a_j^{l-1}$$

Is written as:

$$\delta^l = \left[{w^{l+1}}^T \cdot \delta^{l+1} \right] * \sigma'^l$$
$$b^l = \delta^l$$
$$dw^l = \delta^l \cdot {a^{l-1}}^T$$

During the computation we would like to average over all samples and arrays are stacked column-wise to include the samples.

For example, if δ^l has n rows and the number of samples is m we visualize δ^l as shown below

$$\begin{bmatrix} \delta_{11}^l & \cdots & \delta_{1m}^l \\ \cdots & \cdots & \cdots \\ \delta_{n1}^l & \cdots & \delta_{nm}^l \end{bmatrix}$$

To get the average values we just sum over all the columns and divide by m. This is done in Python by specifying the axis we are summing over.

We get:

$$b^l = \begin{bmatrix} \frac{1}{m}\sum_{s=1}^{m} \delta_{1s}^l \\ \cdots \\ \frac{1}{m}\sum_{s=1}^{m} \delta_{ns}^l \end{bmatrix}$$

The weights are usually a matrix, a result of the tensor product of the deltas and the previous outputs.

So $dw^l = \delta^l . a^{l^T}$ which can be visualized as:

$$\begin{bmatrix} dw_{11}^l & \cdots & dw_{1j}^l \\ \cdots & \cdots & \cdots \\ dw_{i1}^l & \cdots & dw_{ij}^l \end{bmatrix} = \begin{bmatrix} \delta_1^l \\ \cdots \\ \delta_i^l \end{bmatrix} \cdot \begin{bmatrix} a_1^l & \cdots & a_j^l \end{bmatrix}$$

To average over m samples the only extra operation is to divide by m

$$\begin{bmatrix} dw_{11}^l & \cdots & dw_{1j}^l \\ \cdots & \cdots & \cdots \\ dw_{i1}^l & \cdots & dw_{ij}^l \end{bmatrix} = 1/m \begin{bmatrix} \delta_1^l & \cdots & \delta_{1m}^l \\ \cdots & & \\ \delta_i^l & \cdots & \delta_{im}^l \end{bmatrix} \cdot \begin{bmatrix} a_1^l & \cdots & a_j^l \end{bmatrix}$$

A problem encountered during the neural network is called overfitting which usually happens when the NN learns too much details about the training data. Some of the features of the training data are peculiar to it and don’t generalized to other data. Usually in overfitting we get excellent results in the training data but bad results in the testing data.

One way to avoid overfitting is to stop training when the accuracy of the testing data stops improving or even a better technique is to use a validation data set in addition to the training and testing sets.

This strategy of stopping the learning process when the classification accuracy on the validation data has saturated is called early stopping.

The validation data is not only used for early stopping but also to evaluate different values of the parameters such as the number of iterations, learning rate, number of nodes per layer and number of layers.

Previously testing data was used for this purpose but to avoid overfitting we will from now on use the validation data instead which is considered a training data for the hyper-parameters.

But sometimes our data set isn't large enough to be divided into three different data sets and we need this division to avoid overfitting, in this case we can use something called regularization.

Many regularization techniques are available but, in this report, we will be using the L2 regularization.

The basic idea behind this technique is to add a regularization term to the cost function in an L layer neural network as follows:

$$E = -\frac{1}{m}\left[y_i \log a_i^L + (1-y_i)\log(1-a_i^L)\right] + \lambda \sum_w w^2$$

The above equation is the usual cross-entropy discussed earlier plus an added term of the squares of all weights and λ being the regularization parameter.

In general, any cost function can be L2 regularized by adding the square of the weights

$$E = E_0 + \lambda \sum_w w^2$$

This technique was used on the MNIST data set taking only the first 1000 data from the database, using two layers and 30 hidden nodes and a learning rate of 0.5, the accuracy got without regularization was 82% while it was increased to 87% using the regularization. And this value was increased to 98% when the number of hidden layers was increased to 100. Note that with and without regularization the accuracy on the training set was 100%.

By adding the w^2 terms we are making the network prefer smaller weights when all other things are equal. Large weights will be considered only when they considerably improve the first part of the cost function. In that sense the parameter λ controls the relative contribution of the two parts. When the weights are small, a small variation or noise in the input would not make much difference but when the weights are large, even a small variation in the input could lead to significant changes.

Although there is no mathematical proof that regularization makes better prediction and what we have are empirical results that more or less tell us that regularization gives better generalization, some people argue that lower weights reduce complexity of the model (Ocam's razor) but, there is no priori argument on why we should prefer a simpler explanation.

Dropout is another technique used to avoid overfitting and it is the technique used in our code. In this technique and at each iteration randomly half of the hidden nodes are disabled. The remaining ones are used in the forward as well as the backward propagation as shown in the picture below.

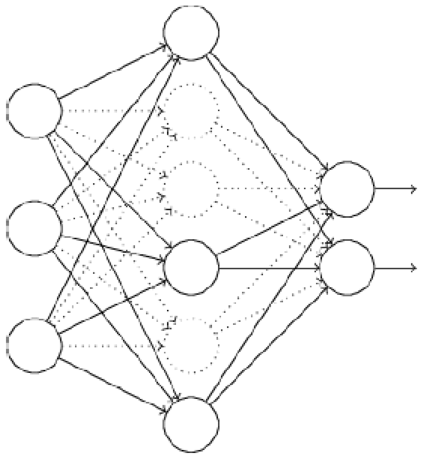


Figure 3.8 Dropout representation

Since at every iteration we are using a different set of nodes to update the parameters, dropout would help regularization. The effect is like having multiple runs over the data or multiple neural networks over the data. The results we obtain is like an average or consensus over those different networks or runs. And by consensus we mean if the majority says it is a 9 then we decide it is a 9.

So far, we initialized the weights randomly, but when we have a large number of inputs the variance for $z_i = \sum w_{ij} x_j$ becomes very broad and there is high probability for large z_i which makes the sigmoid function saturate and therefore learning slowly.

To avoid this problem, we scale all weights by $\sqrt{n_{in}}$ where n_{in} is the number of inputs for a node.

A special kind of neural networks used in this project is called convolution neural networks (CNN).

It is usually used for data that are known to have grid-like topology; examples are time-series data that can be thought of as a 1-dimensional grid of samples taken at regular intervals or image data which can be considered as a 2-dimensional grid of pixels.

CNN have been very successful in practical applications and the name convolution refers to an operation that is similar to the convolution operation used in physics and engineering.

A convolution network usually has at least one convolution layer. A convolution operation is done by multiplying weights element wise with a portion of the input. The same operation with the same weights is repeated over all the inputs. The set of weights are usually referred to as the kernel and the result of the convolution is referred to as a feature map.

The convolution of two functions f and g as used in physics and engineering is defined as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t - \tau) d\tau$$

The discrete version is written as

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m] g[n - m]$$

For finite domains

$$(f * g)[n] = \sum_{m=-M}^{M} f[m] g[n - m]$$

In neural networks we use a related operation, cross-correlation is an operation involving the input I and a kernel K and it is defined as

$$C[i,j] = (I * K)[i,j] = \sum_{m}\sum_{n} I[i+m, j+n]K[m,n]$$

The range of the indices m and n depend on the kernel size. The equation above assumes a stride of size 1 and later we will deal with strides of different sizes.

There are three basic ideas behind convolution: sparse interaction, parameters sharing and equivariant representation.

The NN that we have dealt with so far use matrix multiplication of the input with the weights, each input unit interacts with each output unit.

In convolution each output unit interacts with a portion of the input and this is done by making the kernel small compared with the input which will lead to a smaller number of parameters.

Parameters sharing means that the same weights or kernel are used for all the locations in the input.

A feature available in the convolution network is called equivariance. First, equivariance is not invariance. Let T be the translation operation, f the convolution operation and X the input.

Translation invariance means $f(T(X)) = f(X)$ which is a property not possessed by the convolution operation.

Translation equivariance means $f(T(X)) = T(f(X))$ which is a property possessed by the convolution operation.

What equivariance means is that if a feature is detected by the convolution at position x and reported at output y then when the input is shifted, say by d, the feature will be detected in the output at y + d.

Greyscale example:

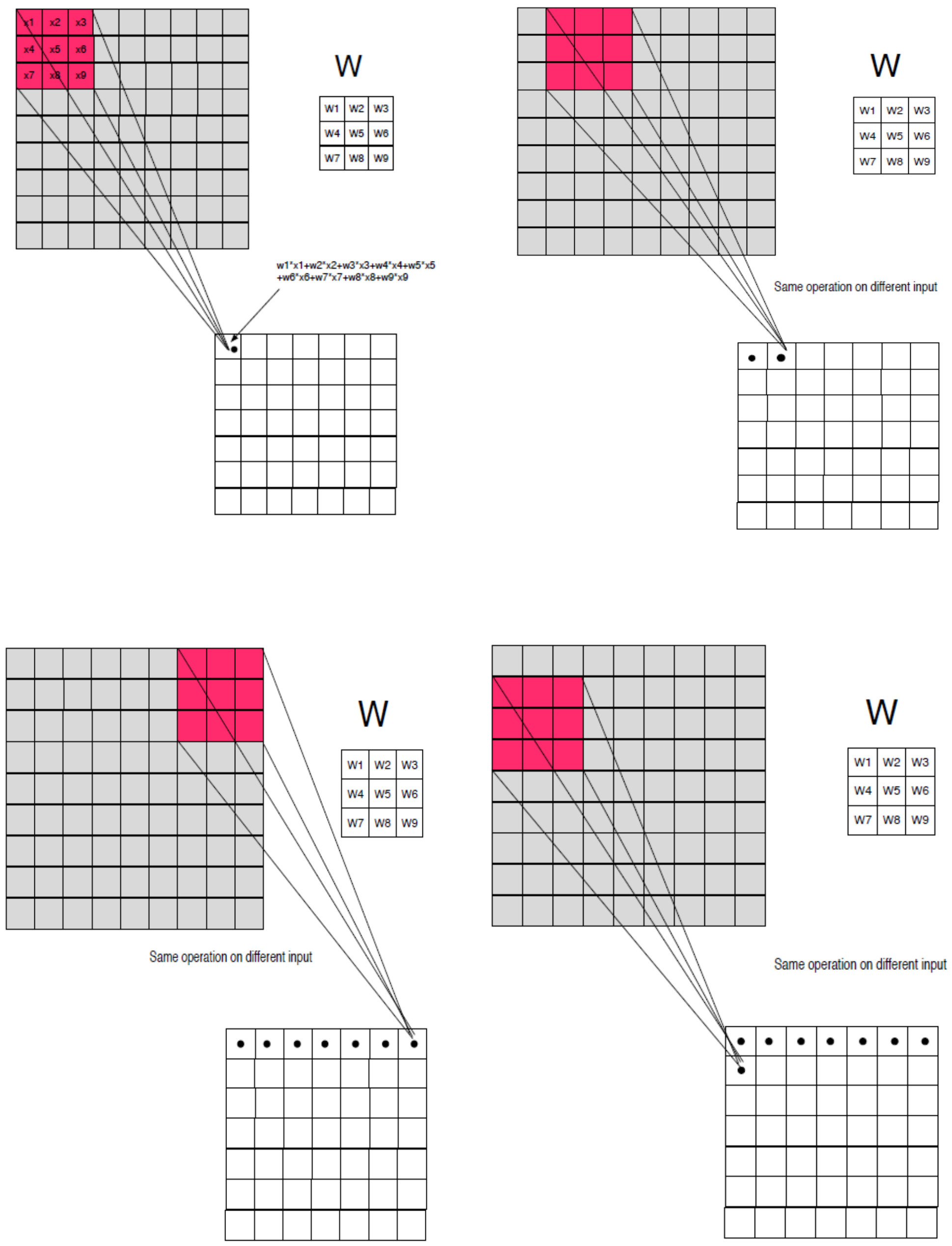

x1 x2 x3
x4 x5 x6
x7 x8 x9
W
W1 W2 W3
W4 W5 W6
W7 W8 W9
w1*x1+w2*x2+w3*x3+w4*x4+w5*x5 +w6*x6+w7*x7+w8*x8+w9*x9
Same operation on different input

Figure 3.9 How convolution works for grayscale images

One technique used in the convolution networks is called pooling and it has two main functions; reduce the spatial size of the input and reduce the amount of parameters.

It is common to insert a pooling layer between successive convolution layers and pooling works independently on every depth-slice of the input and resizes it spatially. Usually, the pooling layer uses filters of size 2 x 2 applied with a stride of 2 and it is done by using the max operation on a receptive field as shown in the figure below.

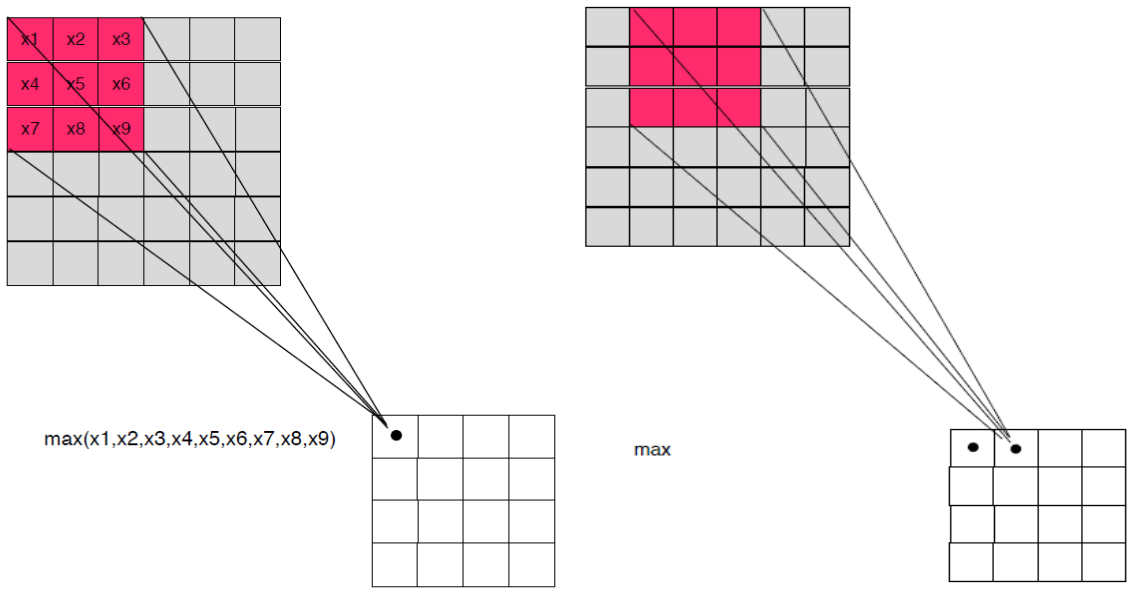


Figure 3.10 Max pooling example

As in the case of convolution if the input is of size W_1 x H_1 x D and we choose the receptive field of size F and a stride S then the output has dimensions W_2 x H_2 x D_2 where:

- $D_2 = D_1$ since the pooling is done per depth slice
- $W_2 = \frac{W_1 - F}{S} + 1$
- $H_2 = \frac{H_1 - F}{S} + 1$

The most common receptive field and stride are $F = 2$ and $S = 2$.

Pooling is used because it replaces a region in the input with a summary statistic usually the max value when using max pooling and this technique makes the computation almost invariant to translation as can be seen in the example below.

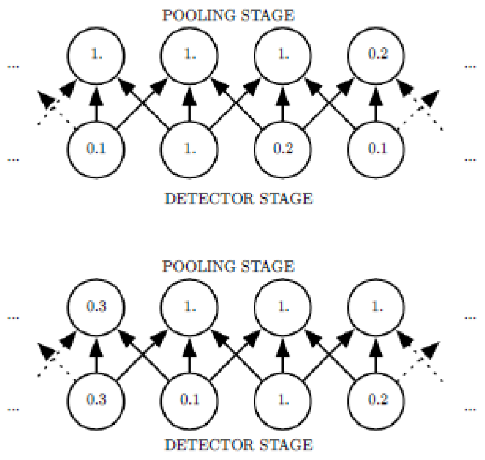


Figure 3.11 Pooling example

The ReLU or rectified linear unit function is used instead of the sigmoid or the softmax and it is defined by $ReLU(x) = \max(0, x)$

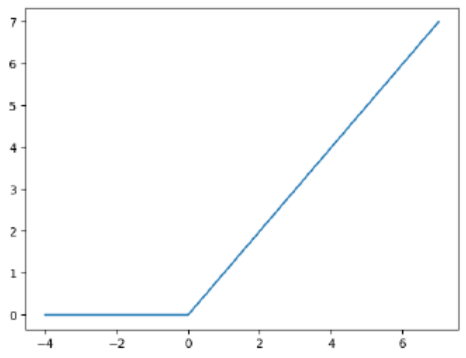


Figure 3.12 Rectified Linear Unit function

The most important advantage of ReLU is the reduced likelihood of vanishing gradient; for large values of the input the sigmoid and its derivative go to zero very quickly which makes learning hard but, in the case of the ReLU function, this does not happen because the derivative is constant.

$\sigma' = \sigma(1 - \sigma) \leq 0.25$ for all values.

Chapter 4: Results

There are many ways to build convolution networks for image recognition. There is always a tradeoff between accuracy and computation time. After many trials, we have found that the best accuracy for a reasonable computation time is achieved using the following deep network architecture.

We used eight layers, five of them convolution while the last three are dense feedforward layers. The architecture of our network is shown in the figure below.

In each convolution layer we have used maxpooling, dropout with a certain rate, and batch normalization. The activation function used is the standard ReLU function.

Our model was implemented using the TensorFlow framework. All runs were performed on Linux system with a NVIDIA Tesla K80 GPU. The approximate running time for each learning run was about 40 minutes.

We have studied the change in prediction accuracy with respect to many parameters. First and foremost we studied the accuracy as a function of the number of training samples. Of a lesser importance is the accuracy as a function of dropout rate and the maxpooling size. We have found that the accuracy dependence on the dropout rate is very small whereas there was a strong dependence on the maxpooling size.

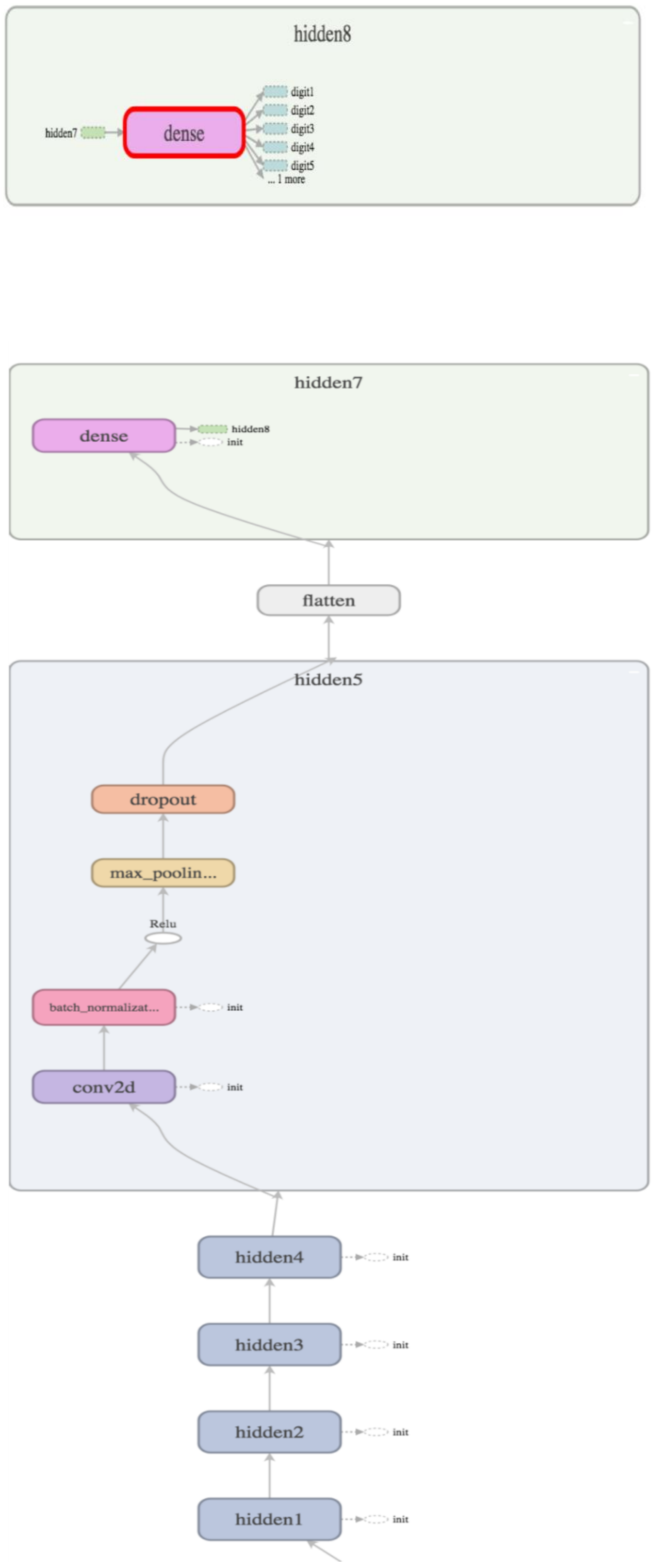


Figure 4.1 Deep network architecture

Our experiments were performed on two types of word CAPTCHAs. The first type involved simple gray scale CAPTCHAs. The second type involved color scale CAPATCHAs with strong distortions. These two types are shown below in Figures 4.2 and 4.3

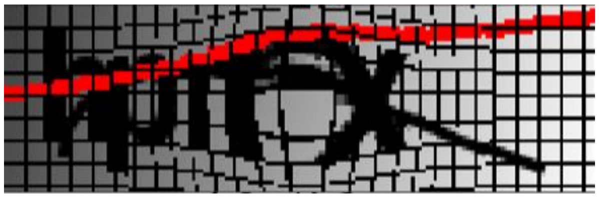


Figure 4.2 Color scale CAPTCHA

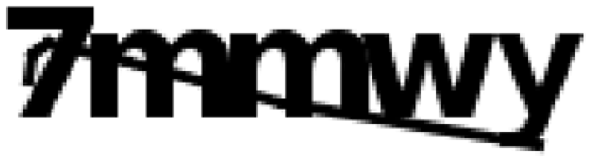


Figure 4.3 Gray scale CAPTCHA

The first results we obtained for different dropout rates and different number of samples but for a fixed 2x2 maxpooling value using color scale and the results are shown in table 4.1 and Figure 4.4 below.

Table 4.1 Dropout rate vs number of samples with 2x2 maxpooling for colored CAPTCHAs

Dropout rate / Number of samples	0.1	0.3	0.6
4000	0.114	0.118	0.138
10000	0.383	0.395	0.415
20000	0.732	0.701	0.706

30000	0.822	0.82	0.814
40000	0.876	0.862	0.877
50000	0.887	0.912	0.904
100000	0.95	0.963	0.968

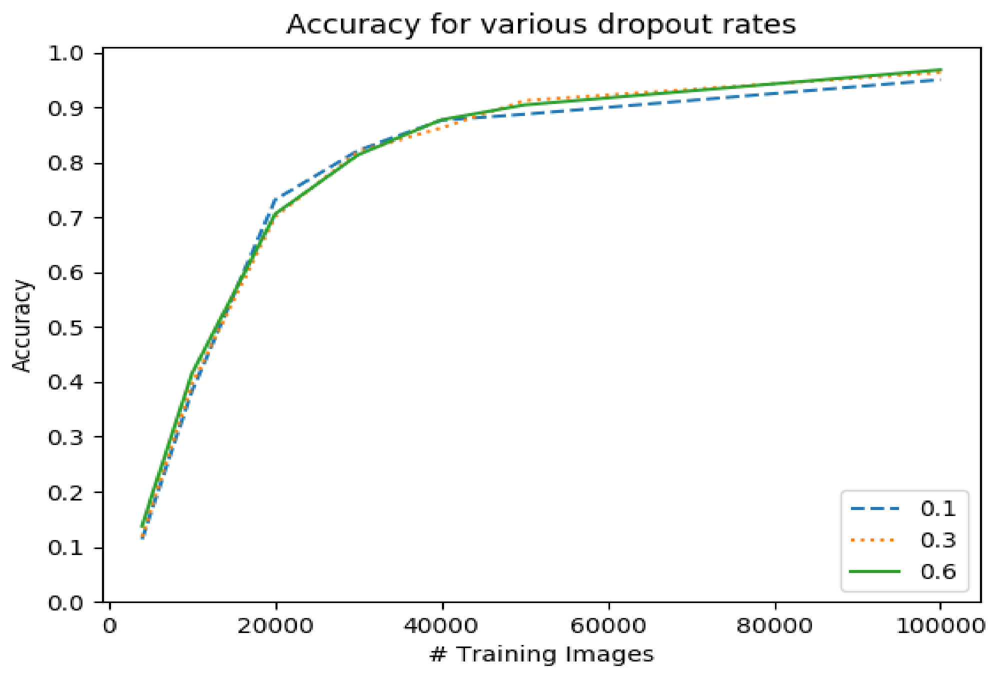


Figure 4.4 Accuracy for various dropout rates (color)

It is very clear that the accuracy for different dropout rates, for the same number of samples, is almost the same.

Having established that the dropout rate has minimal effect on the results we fixed it at 0.1 and we investigated the dependence of the accuracy on the maxpooling size. Table 4.2 and Figure 4.5 below show the results for the same dropout rate fixed to 0.1 and using color scale and different maxpooling values.

Table 4.2 Maxpooling vs. number of samples with 0.1 dropout rate for colored CAPTCHAs

Maxpooling / Number of samples	2x2	4x4	8x8	16x8	16x16
4000	0.114	0.198	0.114	0.006	0.004
10000	0.383	0.675	0.598	0.443	0.185
20000	0.732	0.872	0.867	0.85	0.527
30000	0.822	0.93	0.937	0.909	0.702
40000	0.876	0.949	0.951	0.936	0.785
50000	0.887	0.956	0.968	0.957	0.8
100000	0.95	0.973	0.986	0.966	0.894

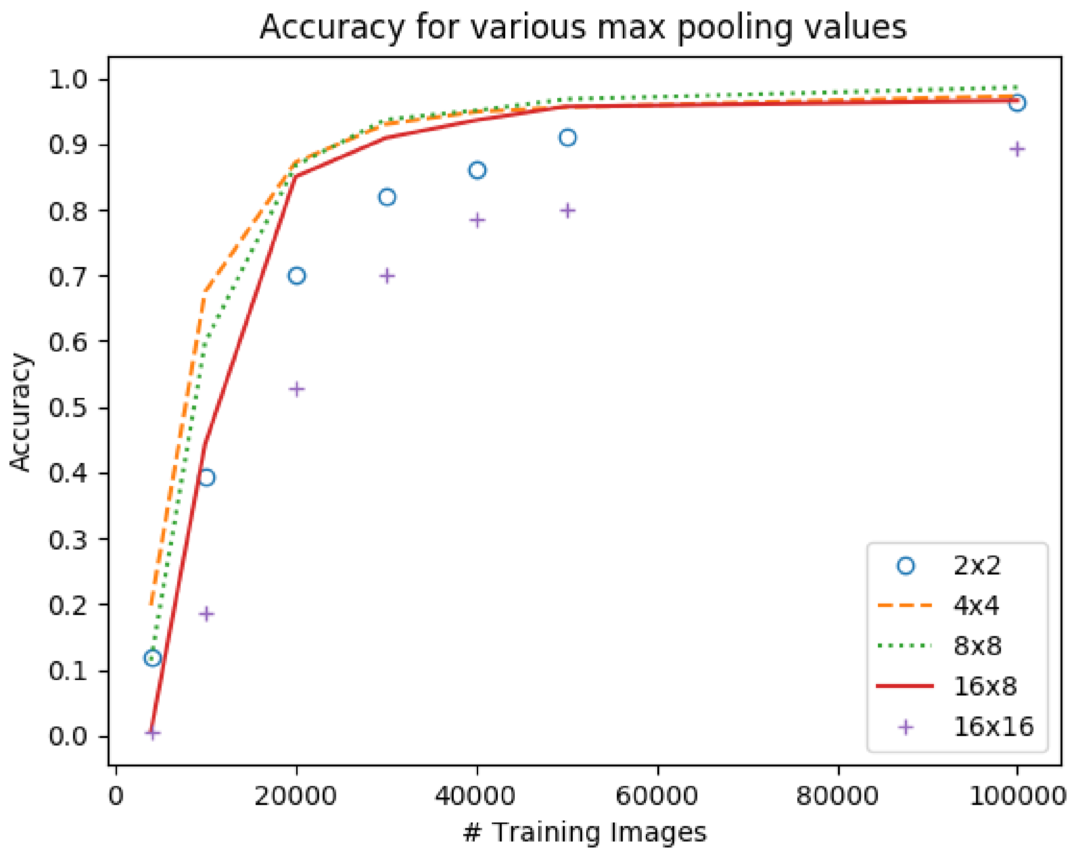


Figure 4.5 Accuracy for various max pooling values (color)

The worst result was obtained for a 16x16 maxpooling value and the best results are for a 4x4 maxpooling and 8x8 maxpooling depending on the number of samples; for a small number of samples 4x4 is found to be better but for higher number of samples, the8x8 maxpooling value gives better accuracy.

Now moving to the gray scale images and for a fixed dropout rate of 0.1 and for different values of maxpooling and number of samples, the results are shown in Table 4.3 and Figure 4.6 below.

Table 4.3 Maxpooling vs. number of samples with 0.1 dropout rate for gray CAPTCHAs

Maxpooling / Number of samples	2x2	4x4	8x8
4000	0.009	0.614	0.346
10000	0.571	0.926	0.832
20000	0.825	0.983	0.979
30000	0.879	0.987	0.983
40000	0.914	0.988	0.993
50000	0.937	0.995	0.993
100000	0.953	1.0	1.0

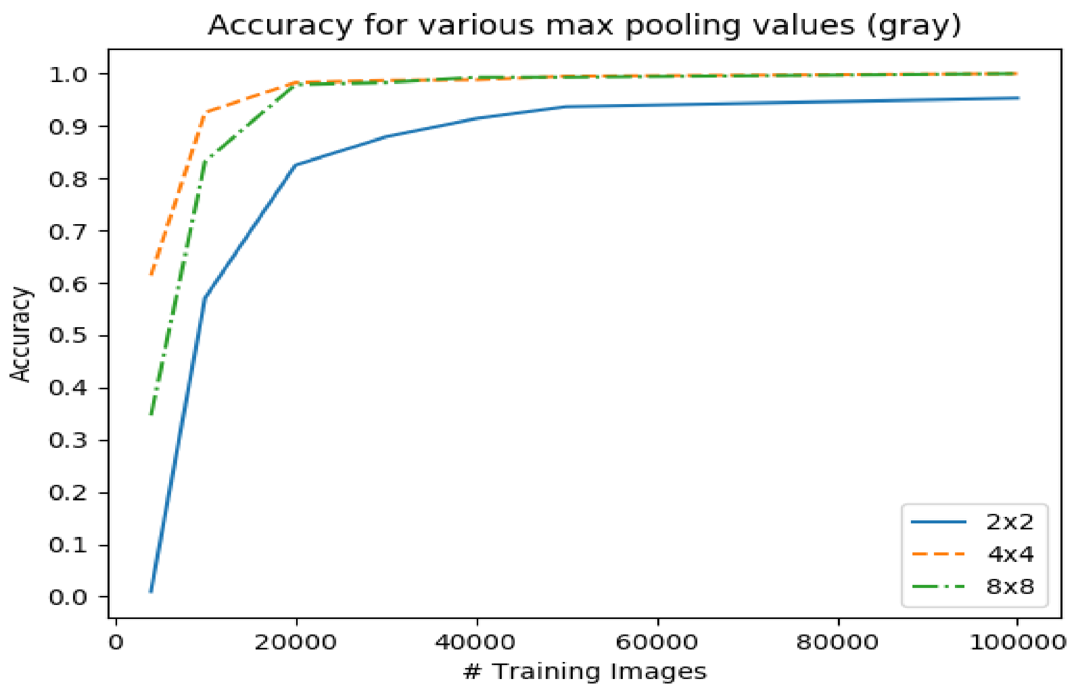


Figure 4.6 Accuracy for various max pooling values (gray)

Obviously 4x4 maxpooling value is giving the highest values of accuracy for different number of samples except for 40000 samples.

Finally, for a fixed maxpooling value of 4x4 and a dropout rate of 0.1 we did a small comparison between color scale and gray scale images and as shown in the graph below gray scale image's experiments where more successful and give a higher accuracy specially for small numbers of samples in my training sets.

Table 4.4 Image type vs number of samples for a 4x4 maxpooling and 0.1 dropout rate

Image / Number of samples	gray	color
4000	0.614	0.198
10000	0.926	0.675
20000	0.983	0.872
30000	0.987	0.93
40000	0.988	0.949
50000	0.995	0.956
100000	1.0	0.973

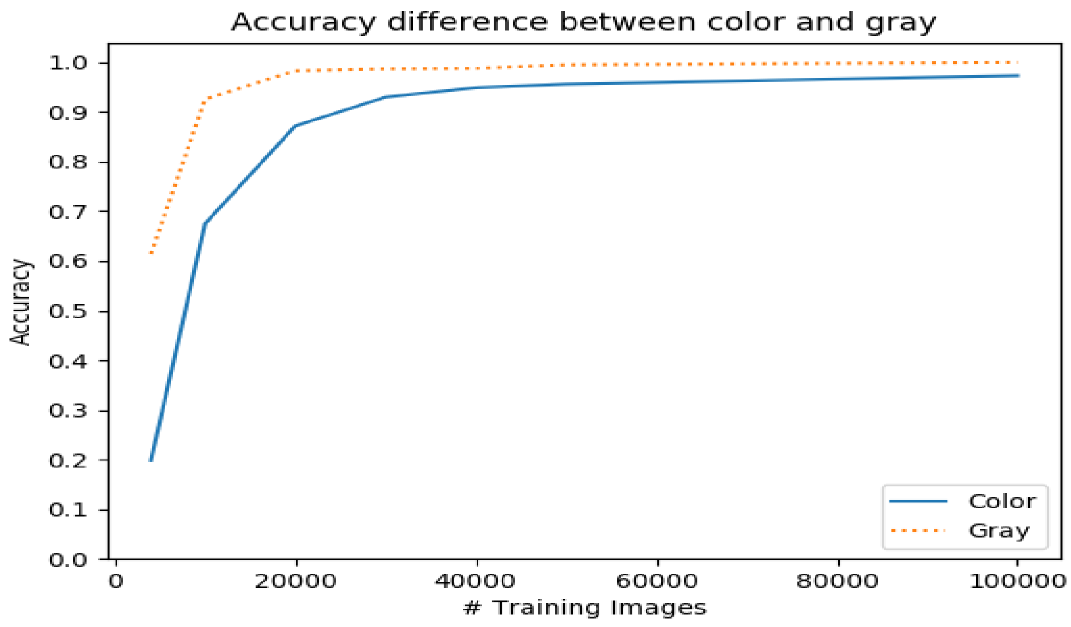


Figure 4.7 Accuracy difference between color and gray

It should be noted that a prediction is deemed to be correct when all characters are predicted correctly. Below we show some failed predictions for both gray scale, Figure 4.8, and color, Figure 4.9, CAPTCHAs. As can be seen all the failure are due to a single character mismatch.

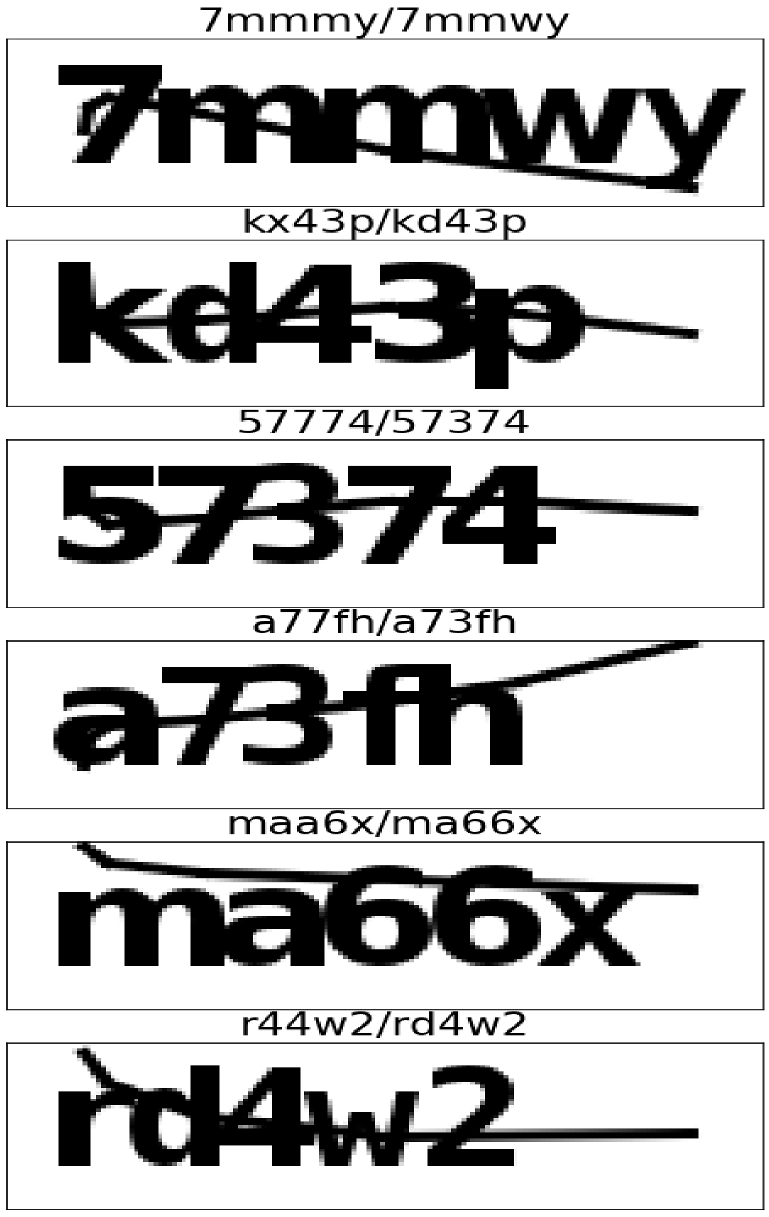


Figure 4.8 Predicted vs labeled gray scale CAPTCHAs

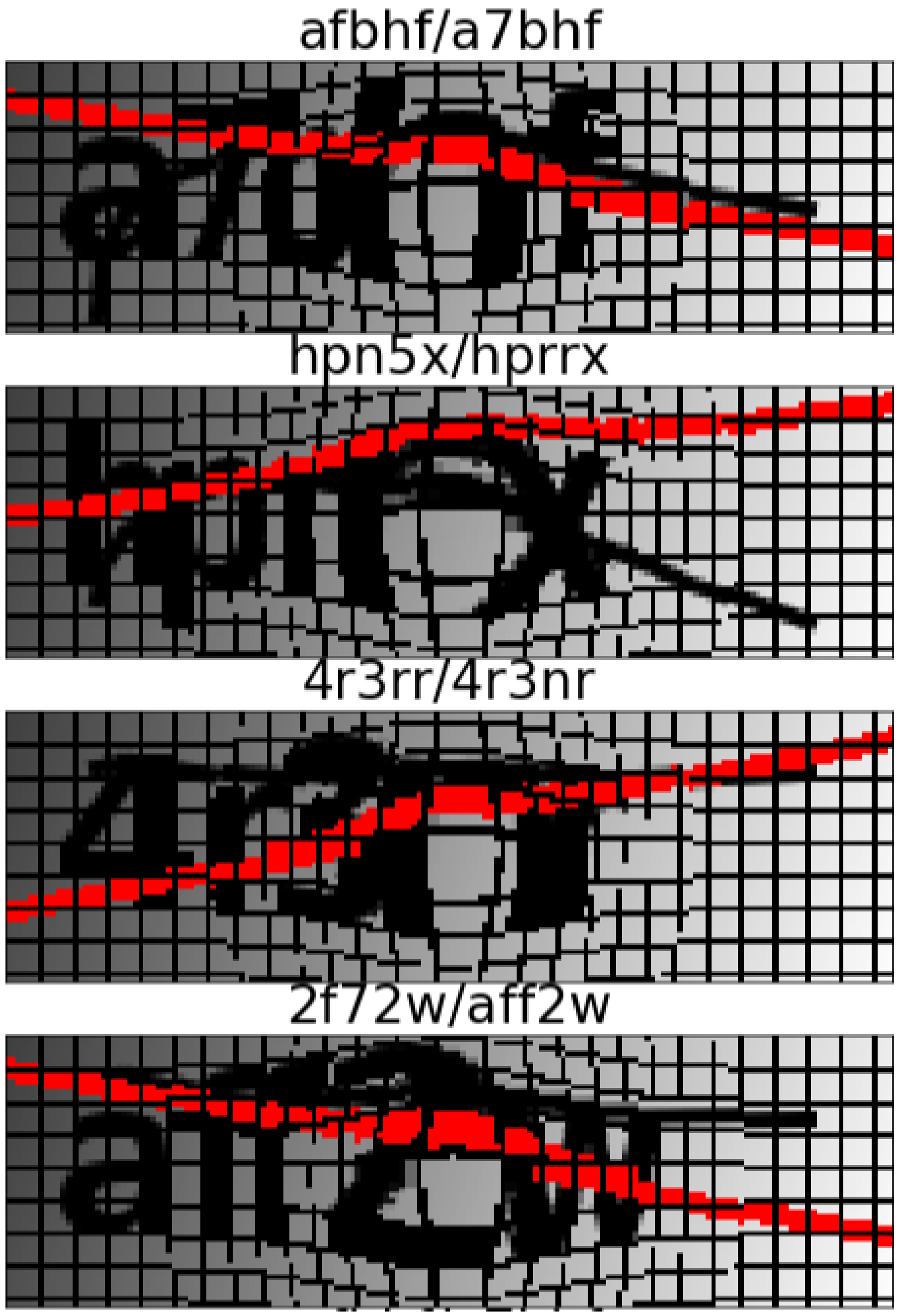


Figure 4.9 Predicted vs labeled color scale CAPTCHAs

Chapter 5: Conclusion

5.1 Summary of the Main Results

In this project, we have tried to break the CAPTCHA using deep neural networks. For machine learning algorithm, a big number of samples is needed so first of all we have found a CAPTCHA generator JAVA script online available in the appendix and used it to generate those needed samples to use in the learning and testing process. Then a machine learning algorithm was used based on convolution networks to teach the algorithm what is the input in a given image, and then testing our results with a data set used for this purpose to get an accuracy. We trained our model for two types of CAPTCHAs color based and gray scale and tried to change some of the parameters used in the algorithm like the maxpooling value and the dropout rate to try to maximize our accuracy on the testing data.

And of course the number of samples is very important in all our experiments, the bigger our dataset used is, the better the accuracy we got.

We found that dropout rate is not very important because the accuracy change was not high but maxpooling value effected more our results and for some number of samples 4x4 maxpooling value was the best but for higher number of samples 8x8 was the optimal choice that got us the highest accuracy which is 98.6% for the colored images.

The same results goes for gray scale images in which dropout was not affecting the results much but maxpooling did and the best accuracy found was 100%.

The training speed for higher number of samples was slower and of course we can’t say that the results found in our project are the best that can one get because we couldn’t test all the possible combinations of parameters used in our code that can be found in the appendix but we can say that the results were satisfying and the accuracy is very good knowing that some of the CAPTCHAs can’t be read even by humans.

5.2 Possible Extensions and Future Work

In this domain, there is always more studies and experiments and machine learning algorithms to do. CAPTCHA creators tend to always find new techniques or new types that can be as easy as possible on humans but hard on the researchers and scientists to break.

Millions of CAPTCHA’s types are available online whenever we surf the web we are being asked to solve a CAPTCHA and as a future work I am thinking of trying to break a new type of these CAPTCHAs

Bibliography

Bishop, C. M. (1995). *Neural networks for pattern recognition*. Oxford university press.
Chow, Y.-W., & Susilo, W. (2017). *Text-based CAPTCHAs over the years*.
Gafni, R., & Nagar, I. (2016). CAPTCHA–Security affecting user experience. *Issues in Informing Science and Information Technology*, *13*, 063–077.
Garg, G., & Pollett, C. (2016). Neural network captcha crackers. *2016 Future Technologies Conference (FTC)*, 853–861. IEEE.
Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.
Hasan, W. K. A. (2016). A Survey of Current Research on Captcha. *International Journal of Computer Science and Engineering Survey (IJCSES)*, *7*(3), 141–157.
Hu, Y., Chen, L., & Cheng, J. (2018). A CAPTCHA recognition technology based on deep learning. *2018 13th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, 617–620. IEEE.
Lei, C. (2015). *Image CAPTCHA technology research based on the mechanism of finger-guessing game*.
Sharma, S., & Seth, N. (2015). Survey of Text CAPTCHA Techniques and Attacks. *Int J Eng Trends Technol*, *22*(6).
Singh, V. P., & Pal, P. (2014). Survey of different types of CAPTCHA. *International Journal of Computer Science and Information Technologies*, *5*(2), 2242–2245.
Soumya, K. R., & Abraham, R. M. (2014). *A Survey on Different CAPTCHA Techniques*.
Von Ahn, L., Blum, M., Hopper, N. J., & Langford, J. (2003). CAPTCHA: Using hard AI problems for security. *International Conference on the Theory and Applications of Cryptographic Techniques*, 294–311. Springer.
Wang, M., Yang, Y., Zhu, M., & Liu, J. (2018). CAPTCHA Identification Based on Convolution Neural Network. *2018 2nd IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, 364–368. IEEE.
Yalamanchili, S., & Rao, M. K. (2011). A framework for devanagari script-based captcha. *ArXiv Preprint ArXiv:1109.0132*.

Appendix A: CODE

Main JAVA:

```
import java.awt.FlowLayout;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.LinkedList;
import java.util.List;

import javax.imageio.ImageIO;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;

import nl.CAPTCHA.CAPTCHA;
import nl.CAPTCHA.CAPTCHA.Builder;
import nl.CAPTCHA.backgrounds.FlatColorBackgroundProducer;
import nl.CAPTCHA.backgrounds.GradiatedBackgroundProducer;
import nl.CAPTCHA.gimpy.DropShadowGimpyRenderer;
import nl.CAPTCHA.gimpy.FishEyeGimpyRenderer;
import nl.CAPTCHA.noise.StraightLineNoiseProducer;
import nl.CAPTCHA.text.producer.ChineseTextProducer;
import nl.CAPTCHA.text.producer.DefaultTextProducer;


public class Main {

	public static void main(String[] args) throws IOException {
		// TODO Auto-generated method stub // Required! Always!
		//img2file("file.png",CAPTCHA.getImage());
		//img2ds("data/train",10000);
		if(args.length!=4){
	    System.out.println("need: dir number width height");
	    System.exit(1);
	  }
	  String dir=args[0];
	  int numOfImages=Integer.parseInt(args[1]);
	  int width=Integer.parseInt(args[2]);
	  int height=Integer.parseInt(args[3]);
		img2ds(dir,numOfImages,width,height);
	}

	public static void img2ds(String dir, int N,int width,int height) throws IOException
	{
		List<String> ans = new LinkedList<String>();
		PrintWriter out = new PrintWriter(dir+"/ans.txt");
		for(int i=1;i<=N;i++)
		{
			if(i%100==0)System.out.println(i+","+N);
			CAPTCHA cap = new CAPTCHA.Builder(width, height)
			    .addText()
			    .addBackground()
			    .addNoise()
			    .gimp()
			    .addBackground(new GradiatedBackgroundProducer())
			    .addNoise(new StraightLineNoiseProducer())
			    .gimp(new FishEyeGimpyRenderer())
			    .build();
			// CAPTCHA cap = new CAPTCHA.Builder(160, 50)
			//      .addText()
			//     //.addBackground(new
FlatColorBackgroundProducer(java.awt.Color.white))
			//     .addBackground()
			//     .addBackground(new GradiatedBackgroundProducer())
			//     .addNoise()
			//     .addNoise(new StraightLineNoiseProducer())
			//      .build();

			img2file(dir+"/"+i+".png",cap.getImage());
			ans.add(cap.getAnswer());
			out.println(cap.getAnswer());
		}
		//System.out.println(ans);
		out.close();
	}

	public static void img2disp(BufferedImage img)
	{

		JFrame frame = new JFrame();
		frame.getContentPane().setLayout(new FlowLayout());
		frame.getContentPane().add(new JLabel(new ImageIcon(img)));
		frame.pack();
		frame.setVisible(true);
	}
	public static void img2file(String fileName, BufferedImage img) throws
IOException
	{
		ImageIO.write(img, "png", new File(fileName));

	}

}
```

First of all we tried to find a dataset of CAPTCHAs with their label online to use it in this study but all the datasets are small and thus not enough for this code to converge so instead we found this java script that was able to generate CAPTCHAs each with its label and we generated 50000 sample for out dataset and saved the result in a datafile.

Decode:

```
import tensorflow as tf

def decode(serialized_example):
  features = tf.parse_single_example(
      serialized_example,
      features={
        'digit1': tf.FixedLenFeature([], tf.int64),
        'digit2': tf.FixedLenFeature([], tf.int64),
        'digit3': tf.FixedLenFeature([], tf.int64),
        'digit4': tf.FixedLenFeature([], tf.int64),
        'digit5': tf.FixedLenFeature([], tf.int64),
        'image_raw': tf.FixedLenFeature([], tf.string)
      })

  image = tf.decode_raw(features['image_raw'], tf.uint8)
# rescale the pixel to 0-1 instead of 0-255. Smaller values lead to better convergence
  image=tf.cast(image,tf.float32)/255.
  image.set_shape((50*160*3))

  # Convert label from a scalar uint64 tensor to an int32 scalar.
#  label = tf.cast(features['label'], tf.int32)
#  label=tf.one_hot(label,10)
# if we use the one_hot encoding then use softmax_cross_entropy instead of
sparse_softmax_cross_entropy
  digit1=features['digit1']
  digit2=features['digit2']
  digit3=features['digit3']
  digit4=features['digit4']
  digit5=features['digit5']
  return image,tf.stack([digit1,digit2,digit3,digit4,digit5])
```

Because the previous code was generating images of 5 digits that can be a letter or a number, in this code we are able to separate each digit alone and work on it separately.

Convert tf records:

```
#convert the mnist data from a pickle file to tfrecords file
import argparse
import numpy as np
import gzip
import cPickle
import random
import tensorflow as tf
from PIL import Image
import matplotlib.pyplot as plt


def _int64_feature(value):
  return tf.train.Feature(int64_list=tf.train.Int64List(value=[value]))


def _bytes_feature(value):
  return tf.train.Feature(bytes_list=tf.train.BytesList(value=[value]))

def char_code(label):

  ascii=ord(label)
  if ascii > 57:
    v=ascii-87
  else:
    v=ascii-48


  return v

parser=argparse.ArgumentParser()
parser.add_argument('--data_dir',required=True)
parser.add_argument('--record_file',required=True)
args=parser.parse_args()
data_dir=args.data_dir
record_file=data_dir+"/"+args.record_file


def load_dataset(record_file):
  file_list=[]
  X=[]
  Y=[]
  with open(data_dir+"ans.txt") as f:
    lines=f.readlines()
    lines=map(lambda x:x.strip(),lines)
  list_of_files=tf.gfile.Glob(data_dir+"*.png")
  with tf.python_io.TFRecordWriter(record_file) as writer:
    for index, file in enumerate(list_of_files):
      tr=file.split('/')
      i=int(tr[-1].split('.')[0])
      c=lines[i-1]
      img=Image.open(file)
      image_raw=np.uint8(img)
      image_raw=image_raw.reshape(50*160*3)
      image_raw=image_raw.tostring()
      example = tf.train.Example(
        features=tf.train.Features(
          feature={
            'digit1': _int64_feature(char_code(c[0])),
            'digit2': _int64_feature(char_code(c[1])),
            'digit3': _int64_feature(char_code(c[2])),
            'digit4': _int64_feature(char_code(c[3])),
            'digit5': _int64_feature(char_code(c[4])),
            'image_raw': _bytes_feature(image_raw)
          }))
      writer.write(example.SerializeToString())

init = tf.global_variables_initializer()
load_dataset(record_file)
```

In this code our digits are converted to records that tensor flow understands. It downloads the records from the previous code that has been saved to a file named record_file and it reshapes the images to a vector of size 50*160*3 and saves the answer into the same file.

Train:

```
import numpy as np
import argparse
import tensorflow as tf
from model import inference
from loss import loss
from decode import decode

base_rate = 0.01
decay_steps=1000
decay_rate=0.9
training_epochs = 20
batch_size = 50
dropout_rate=0.6
```

```
parser=argparse.ArgumentParser()
parser.add_argument('--path_to_tfrecords_file',required=True)
parser.add_argument('--path_to_checkpoint_file',required=True)
parser.add_argument('--restart',dest='restart',action='store_true')
parser.add_argument('--no-restart',dest='restart',action='store_false')
args=parser.parse_args()

data_path=args.path_to_tfrecords_file
path_to_checkpoint=args.path_to_checkpoint_file+"/latest.ckpt"
restart=args.restart
#data_path='data/train/train.tfrecords'
#path_to_checkpoint='sg/latest.ckpt'

num_records=sum(1 for _ in tf.python_io.tf_record_iterator(data_path))


filename_queue=tf.train.string_input_producer([data_path],num_epochs=None)
reader=tf.TFRecordReader()
_,serialized_example=reader.read(filename_queue)

image,y=decode(serialized_example)

X,Y=tf.train.shuffle_batch([image,y],batch_size=batch_size,capacity=10000,num_threads=
4,min_after_dequeue=9999)

logits=inference(X,dropout_rate)
loss_op=loss(logits,Y)
step_counter = tf.Variable(0, name='step_counter', trainable=False)
learning_rate = tf.train.exponential_decay(base_rate, global_step=step_counter,
                                           decay_steps=decay_steps, decay_rate=decay_rate,
staircase=True)
optimizer=tf.train.GradientDescentOptimizer(learning_rate=learning_rate)

train_op=optimizer.minimize(loss_op,global_step=step_counter)

init = tf.global_variables_initializer()
print "number of records is {}".format(num_records)

with tf.Session() as sess:
   sess.run(init)
   coord = tf.train.Coordinator()
   threads = tf.train.start_queue_runners(coord=coord)
   saver=tf.train.Saver()
   if not restart:
      saver.restore(sess,path_to_checkpoint)

   num_batches=num_records/batch_size
```

```
  for epoch in range(training_epochs):
    av=0
    for batch in range(num_batches):
      _,c,r=sess.run([train_op,loss_op,learning_rate])
      if batch%100==0:
          print c,r
      av+=c
    print (" epoch {}: cost={} ".format(epoch,av/(num_batches)))


  saver.save(sess,path_to_checkpoint)

  x,y,pred=sess.run([X,Y,logits])
  coord.request_stop()
  coord.join(threads)
```

In this part we are training our set with all the arguments set as shown in the code above.

Model:

```
import numpy as np
import tensorflow as tf

def inference(x, drop_rate):
        x=tf.reshape(x,shape=[-1,50,160,3])
#input (50,160)
        with tf.variable_scope('hidden1'):
            conv = tf.layers.conv2d(x, filters=48, kernel_size=[5, 5], padding='same')
            norm = tf.layers.batch_normalization(conv)
            activation = tf.nn.relu(norm)
            pool = tf.layers.max_pooling2d(activation, pool_size=[2, 2], strides=2,
padding='same')
            dropout = tf.layers.dropout(pool, rate=drop_rate)
            hidden1 = dropout
#input (25,80)
        with tf.variable_scope('hidden2'):
            conv = tf.layers.conv2d(hidden1, filters=128, kernel_size=[5, 5], padding='same')
            norm = tf.layers.batch_normalization(conv)
            activation = tf.nn.relu(norm)
            pool = tf.layers.max_pooling2d(activation, pool_size=[2, 2], strides=2,
padding='same')
            dropout = tf.layers.dropout(pool, rate=drop_rate)
            hidden2 = dropout
#input(13,40)
        with tf.variable_scope('hidden3'):
            conv = tf.layers.conv2d(hidden2, filters=160, kernel_size=[5, 5], padding='same')
            norm = tf.layers.batch_normalization(conv)
            activation = tf.nn.relu(norm)
            pool = tf.layers.max_pooling2d(activation, pool_size=[2, 2], strides=1,
padding='same')
            dropout = tf.layers.dropout(pool, rate=drop_rate)
            hidden3 = dropout
#input(13,40)
        with tf.variable_scope('hidden4'):
            conv = tf.layers.conv2d(hidden3, filters=192, kernel_size=[5, 5], padding='same')
            norm = tf.layers.batch_normalization(conv)
            activation = tf.nn.relu(norm)
            pool = tf.layers.max_pooling2d(activation, pool_size=[2, 2], strides=2,
padding='same')
            dropout = tf.layers.dropout(pool, rate=drop_rate)
            hidden4 = dropout

#input(7,20)
        with tf.variable_scope('hidden5'):
            conv = tf.layers.conv2d(hidden4, filters=192, kernel_size=[5, 5], padding='same')
            norm = tf.layers.batch_normalization(conv)
            activation = tf.nn.relu(norm)
            pool = tf.layers.max_pooling2d(activation, pool_size=[2, 2], strides=2,
padding='same')
            dropout = tf.layers.dropout(pool, rate=drop_rate)
            hidden5 = dropout
#input (4,10)


        flatten = tf.contrib.layers.flatten(hidden5)
#input 4x10x192=7680
        with tf.variable_scope('hidden7'):
            dense = tf.layers.dense(flatten, units=7680, activation=tf.nn.relu)
            hidden7 = dense

        with tf.variable_scope('hidden8'):
            dense = tf.layers.dense(hidden7, units=7680, activation=tf.nn.relu)
            hidden8 = dense


        with tf.variable_scope('digit1'):
            dense = tf.layers.dense(hidden8, units=36)
            digit1 = dense

        with tf.variable_scope('digit2'):
            dense = tf.layers.dense(hidden8, units=36)
            digit2 = dense
        with tf.variable_scope('digit3'):
            dense = tf.layers.dense(hidden8, units=36)
            digit3 = dense

        with tf.variable_scope('digit4'):
            dense = tf.layers.dense(hidden8, units=36)
            digit4 = dense

        with tf.variable_scope('digit5'):
            dense = tf.layers.dense(hidden8, units=36)
            digit5 = dense

        logits = tf.stack([digit1,digit2,digit3,digit4,digit5],axis=1)
        return logits
```

Loss:

```
import tensorflow as tf

def loss(logits, digits_labels):
        digit1_cross_entropy =
tf.reduce_mean(tf.losses.sparse_softmax_cross_entropy(labels=digits_labels[:, 0],
logits=logits[:,0]))
        digit2_cross_entropy =
tf.reduce_mean(tf.losses.sparse_softmax_cross_entropy(labels=digits_labels[:, 1],
logits=logits[:,1]))
        digit3_cross_entropy =
tf.reduce_mean(tf.losses.sparse_softmax_cross_entropy(labels=digits_labels[:, 2],
logits=logits[:,2]))
        digit4_cross_entropy =
tf.reduce_mean(tf.losses.sparse_softmax_cross_entropy(labels=digits_labels[:, 3],
logits=logits[:,3]))
        digit5_cross_entropy =
tf.reduce_mean(tf.losses.sparse_softmax_cross_entropy(labels=digits_labels[:, 4],
logits=logits[:,4]))
        loss =  digit1_cross_entropy + digit2_cross_entropy + digit3_cross_entropy +
digit4_cross_entropy + digit5_cross_entropy
        return loss
```

Test:

```
import numpy as np
import argparse
import tensorflow as tf
import os
from model import inference
from decode import decode

os.environ["CUDA_VISIBLE_DEVICES"]="-1"
batch_size = 1000
```

```
parser=argparse.ArgumentParser()
parser.add_argument('--path_to_tfrecords_file',required=True)
parser.add_argument('--path_to_checkpoint_file',required=True)
args=parser.parse_args()

def char_code(value):

  if value >9:
    v=value+87
  else:
    v=value+48


  return chr(v)


data_path=args.path_to_tfrecords_file
path_to_checkpoint=args.path_to_checkpoint_file+"/latest.ckpt"
#path_to_checkpoint="model-small/latest.ckpt"
#data_path='data-small/test/test.tfrecords'

num_records=sum(1 for _ in tf.python_io.tf_record_iterator(data_path))
batch_size=num_records

filename_queue=tf.train.string_input_producer([data_path],num_epochs=None)
reader=tf.TFRecordReader()
_,serialized_example=reader.read(filename_queue)

image,y=decode(serialized_example)
X,Y=tf.train.batch([image,y],batch_size=batch_size)

logits=inference(X,0.0)

# Initializing the variables
init = tf.global_variables_initializer()




with tf.Session() as sess:
    sess.run(init)
    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(coord=coord)
    restorer=tf.train.Saver()
    restorer.restore(sess,path_to_checkpoint)
    x,y,pred=sess.run([X,Y,tf.nn.softmax(logits)])


    coord.request_stop()
    coord.join(threads)


res=np.argmax(pred,axis=2)
nomatch=np.array(list(set(np.where(y-res!=0)[0])))
print(' correct {} out of {} for accuracy of {}'.format(x.shape[0]-
nomatch.size,x.shape[0],float((x.shape[0]-nomatch.size))/x.shape[0]))
pairs=[]
for i in nomatch:
  pairs.append(("".join(map(char_code,res[i])),x[i]))
```