Chapter-1

INTRODUCTION

1. INTRODUCTION

Signature is a socially accepted and extensively used for identification of an Individual. It has an assumption that signature changes slowly and is virtually impossible to forge without detection. Increasing identification requirements and security paradigm shift have placed biometrics and particularly Signature detection and verification at the center of much attention. The term biometrics refers to individual recognition based on a person's distinguishing characteristics. It has an advantage over token based approach of not being lost and over knowledge based approach of not being forgotten. Thereby biometrics can be applied to several biometric modalities like iris, fingerprint, face, vein etc. However handwritten signatures are recognized to be one of the most common techniques in establishing the identity of an individual.

A good signature verification system has wide range of applications in diverse fields which include access control system, electronic fund transfer, bank operation, document analysis etc.

1.1 MAIN FEATURES OF A SIGNATURE

1. Global Features: These are extracted from the whole signature, including block codes, Wavelet and Fourier transforms. They can be extracted easily and are tough to noise. However they only deliver limited information for signature verification.
2. Local Features: These are calculated to describe geometrical characteristics such as location, tangent track and curving. Local features provide affluent features of writing shapes and are powerful for cultivated writers.. Having said that however extraction of consistent local features is still a hard problem.

1.2. DIFFERENT KINDS OF FORGERIES

The objective of signature verification system is to discriminate between two signature classes, the genuine and fake signatures. There are 3 different kinds of forgeries we generally see

1. Random Forgery: In these kinds of forgeries the person has no idea of a person's name or the style of signature.
2. Simple forgery: These kinds of forgeries simply include imitating someone else's signature with only decent knowledge of its local features.

3. Skilled Forgeries: This is generally signed by a person who has access to a genuine signature for practice.

With this project we aim to create a system which increases the efficiency of classifying signatures.

1.3 CHALLENGES

Challenges leading to the practical applications of Forgery Detection require us to cross a big hurdle of data being available. For any machine learning system to work better, we require a data to train the system. So how much data should be available? The answer would be the more the better. And that stands as a problem. In general for every class of signature you have a few samples available which maybe 3-4 at max.

Our project, apart from the problem of forgery detection also solves the problem of data by incorporating 'Few Shot Learning' methods. A newly designed method which is supervised but can find out differences using very few samples of training data

Instead of learning all the differences between the pictures we would like to find out the difference between them which means between their features and conclude accordingly.

1.4 WHAT I INTEND TO DO?

Machine learning has been successfully used to achieve state-of- the-art performance in a variety of applications such as web search, spam detection, caption generation, and speech and image recognition. However, these algorithms often break down when forced to make predictions about data for which little supervised information is available. We desire to generalize to these unfamiliar categories without necessitating extensive retraining which may be either expensive or impossible due to limited data. We intend to use few shot classification to make predictions and generate outcomes. This will help us to attain high accuracy with a reasonably small training set.

Chapter-2

LITERATURE SURVEY

2. LITERATURE SURVEY

The process of learning good features for machine learning applications can be very computationally expensive and may prove difficult in cases where little data is available. A prototypical example of this is the few-shot learning set-ting, in which we must correctly make predictions given only a single example of each new class. In this paper, we explore a method for learning siamese neural networks which employ a unique structure to naturally rank similarity between inputs. Once a network has been tuned, we can then capitalize on powerful discriminative features to generalize the predictive power of the network not just to new data, but to entirely new classes from unknown distributions. Using a convolutional architecture, we are able to achieve strong results which exceed those of other deep learning models with near state-of-the-art performance on few-shot classification tasks.

Humans exhibit a strong ability to acquire and recognize new patterns. In particular, we observe that when presented with stimuli, people seem to be able to understand new concepts quickly and then recognize variations on these concepts in future percepts. Machine learning has been successfully used to achieve state-of-the-art performance in a variety of applications such as web search, spam detection, caption generation, and speech and image recognition. However, these algorithms often break down when forced to make predictions about data for which little supervised information is available. We desire to generalize to these unfamiliar categories without necessitating extensive retraining which may be either expensive or impossible due to limited data or in an online prediction setting, such as web retrieval.

One particularly interesting task is classification under the restriction that we may only observe a single example of each possible class before making a prediction about a test instance. This is called few-shot learning and it is the primary focus of our model presented in this work This should be distinguished from zero-shot learning, in which the model cannot look at any examples from the target classes.

Few-shot learning can be directly addressed by developing domain-specific features or inference procedures which possess highly discriminative properties for the target task. As a result, systems which incorporate these methods tend to excel at similar instances but fail to offer robust solutions that may be applied to other types of problems. In this pa-per, we present a novel approach which limits

assumptions on the structure of the inputs while automatically acquiring features which enable the model to generalize success-fully from few examples. We build upon the deep learning framework, which uses many layers of non-linearity to capture invariance's to transformation in the input space, usually by leveraging a model with many parameters and then using a large amount of data to prevent over fitting. These features are very powerful because we are able to learn them without imposing strong priors, although the cost of the learning algorithm itself may be considerable.

2.1. Approach

In general, we learn image representations via a supervised metric-based approach with siamese neural networks, and then reuse that network's features for few-shot learning without any retraining.

For this domain, we employ large siamese convolutional neural networks which are capable of learning generic image features useful for making predictions about unknown class distributions even when very few examples from these new distributions are available; are easily trained using standard optimization techniques on pairs sampled from the source data; and provide a competitive approach that does not rely upon domain-specific knowledge by instead exploiting deep learning techniques.

To develop a model for few-shot image classification, we aim to first learn a neural network that can discriminate between the class-identity of image pairs, which is the standard verification task for image recognition. We hypothesize that networks which do well at verification should generalize to few-shot classification. The verification model learns to identify input pairs according to the probability that they belong to the same class or different classes. This model can then be used to evaluate new images, exactly one per novel class, in a pairwise manner against the test image. The pairing with the highest score according to the verification network is then awarded the highest probability for the few-shot task. If the features learned by the verification model are sufficient to confirm or deny the identity of characters from one set of alpha-bets, then they ought to be sufficient for other alphabets, provided that the model has been exposed to a variety of alphabets to encourage variance amongst the learned features.

2.2 Related Work

Overall, research into few-shot learning algorithms is fairly immature and has received limited attention by the machine learning community. There are nevertheless a few key lines of work which precede this paper.

The seminal work towards few-shot learning dates back to the early 2000's with work by Li Fei-Fei et al. The authors developed a variational Bayesian framework for few-shot image classification using the premise that previously learned classes can be leveraged to help forecast future ones when very few examples are available from a given class . More recently, Lake et al. approached the problem of few-shot learning from the point of view of cognitive science, ad-dressing few-shot learning for character recognition with a method called Hierarchical Bayesian Program Learning (HBPL) In a series of several papers, the authors modeled the process of drawing characters generatively to decompose the image into small pieces. The goal of HBPL is to determine a structural explanation for the observed pixels. However, inference under HBPL is difficult since the joint parameter space is very large, leading to an intractable integration problem.

Some researchers have considered other modalities or transfer learning approaches. Lake et al. have some very recent work which uses a generative Hierarchical Hid-den Markov model for speech primitives combined with a Bayesian inference procedure to recognize new words by unknown speakers. Maas and Kemp have some of the only published work using Bayesian networks to predict attributes for Ellis Island passenger data. Wu and Dennis address few-shot learning in the context of path planning algorithms for robotic actuation . Lim focuses on how to "borrow" examples from other classes in the training set by adapting a measure of how much each category should be weighted by each training exemplar in the loss function

Chapter3- PROBLEM DEFINITION AND FUNCTIONALITY

3.1 PROBLEM DEFINITION

Before we try to solve any problem, we should first precisely state what the problem actually is, so here is the problem of Signature Forgery Detection using few-shot classification expressed symbolically:

Our model is given a tiny labeled training set S, which has N examples, each vectors of the same dimension with a distinct label y.

$S=\{(x1,y1),\ldots,(xN,yN)\}$

It is also given x` the test example it has to classify. Since only few example in the support set has the right class, the aim is to correctly predict which $y \in S$ is the same as x` label, y and then classify the data

3.2THINGS WE CONSIDER

There are fancier ways of defining the problem, but this one is ours. Here are some things to make note of:

- Real world problems might not always have the constraint that exactly one image has the correct class
- It's easy to generalize this to k-shot learning by having there be k examples for each y than just one.
- When N is higher, there are more possible classes that x` can belong to, .so it's harder to predict the correct one.
- Random guessing will average (100/n) % accuracy.

If we naively train a neural network on a few-shot as a vanilla cross-entropy-loss softmax classifier, it will severely over fit. Heck, even if it was a *hundred* shot learning a modern neural net would still probably over fit. Big neural networks have millions of parameters to adjust to their data and so they can learn a huge space of possible functions. (More formally, they have a high VC Dimension, which is part of why they do so well at learning from complex data with high dimensionality.) Unfortunately this strength also appears to be their undoing for few-shot learning. When there are millions of parameters to gradient descend upon, and a staggeringly huge number of possible mappings that can be

learned, how can we make a network learn one that generalizes when there's just a single example to learn from?

It's easier for humans to few-shot learn the concept of a spatula or the letter Θ because they have spent a lifetime observing and learning from similar objects. It's not really fair to compare the performance of a human who's spent a lifetime having to classify objects and symbols with that of a randomly initialized neural net, which imposes a very weak prior about the structure of the mapping to be learned from the data. This is why most of the few-shot learning papers I've seen take the approach of *knowledge transfer* from other tasks.

Neural nets are really good at extracting useful features from structurally complex/high dimensional data, such as images. If a neural network is given training data that is similar to (but not the same as) that in the few-shot task, it might be able to learn useful features which can be used in a simple learning algorithm that doesn't require adjusting these parameters. It still counts as few-shot learning as long as the training examples are of different classes to the examples used for few-shot testing.

(NOTE: Here a *feature* means a "transformation of the data that is useful for learning".)

So now an interesting problem is *how do we get a neural network to learn the features?* The most obvious way of doing this (if there's labeled data) is just vanilla transfer learning - train a softmax classifier on the training set, then fine-tune the weights of the last layer on the support set of the few-shot task. In practice, neural net classifiers don't work too well for data like omniglot where there are few examples per class, and even fine tuning only the weights in the last layer is enough to over fit the support set. Still works quite a lot better than L2 distance nearest neighbor though table of various deep few-shot learning methods and their accuracy.)

There's a better way of doing it though! Remember 1 nearest neighbor? This simple, non-parametric few-shot learner just classifies the test example with the same class of whatever support example is the closest in L2 distance. This works ok, but L2 Distance suffer from the ominous sounding curse of dimensionality and so won't work well for data with thousands of dimensions like omniglot. Also, if you have two nearly identical images and move one over a few pixels to the right the L2 distance can go from being almost zero to being really high. L2 distance is a metric that is just woefully inadequate

for this task. Deep learning to the rescue? We can use a deep convolution network to learn some kind of similarity function that a non-parametric classifier like nearest neighbor can use.

As discussed our network model will consists of 2 CNN's with which the weights and biases for the network would be divided.

So we first discuss about what is a CNN and then we would describe the discriminator network model and then we would go on to describe the network model all together

3.3 Convolution Neural Networks

When you first heard of the term convolution neural networks, you may have thought of something related to neuroscience or biology, and you would be right. Sort of. CNNs do take 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. 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.

Back to the specifics. A more detailed overview of what CNNs do would be that you take the image, pass it through a series of convolution, 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. So let's get into the most important one.

The first layer in a CNN is always a **Convolution Layer**. First thing to make sure you remember is what the input to this conv (I'll be using that abbreviation a lot) 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 conv 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 dimension 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 you 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.

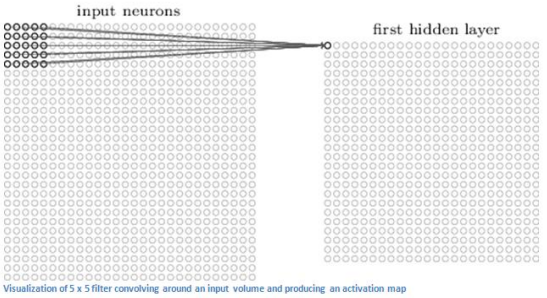
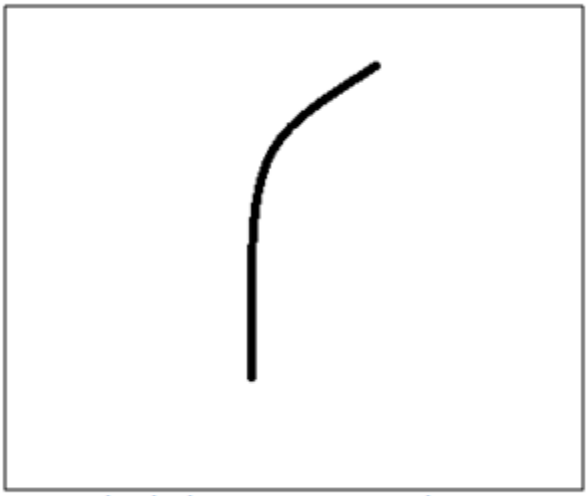


Fig 3.1 Visualization of filter and receptive field

However, 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**. When I say features, I'm talking about 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



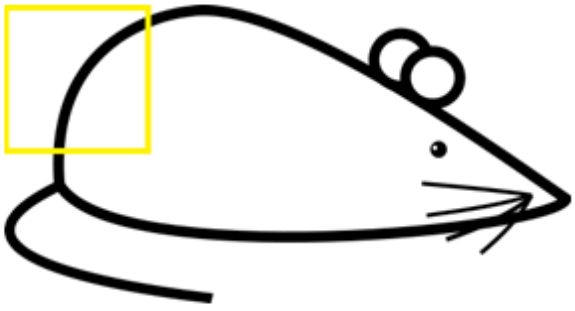
Visualization of a curve detector filter

Fig. 2.1- Matrix value of filter field

Fig. 2.2- Visualization of the filter

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.

Original image



Visualization of the filter on the image

Fig. 2.3- Original Image

Fig. 2.4- Visualization of the filter

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

Visualization of the receptive field

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!)

Fig.2.5 simulation of matrix convolution on sample data

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.

Visualization of the filter on the image

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

Fig. 2.6 simulation of matrix convolution on sample data

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 conv 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 26 x 26 x 1 activation map (26 because of the 7x7 filter instead of 5x5) 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.

Now, this is the one aspect of neural networks that I purposely haven't mentioned yet and it is probably the most important part. There may be a lot of questions you had while reading. 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. Back to backprop.

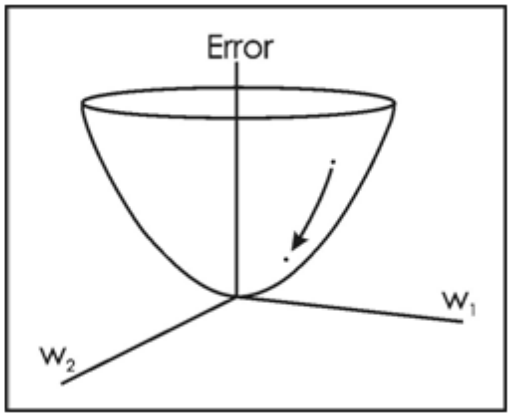
So backpropagation can be separated into 4 distinct sections, the forward pass, the loss function, the backward pass, and the weight update. During the **forward pass**, you take a training image which as we remember is a 32 x 32 x 3 array of numbers and pass it through the whole network. On our first 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.

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

Eq.1 Loss function

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 ConvNet) 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.

Fig 2.7 Visualization of loss function

This is the mathematical equivalent of a **dL/dW** where W 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

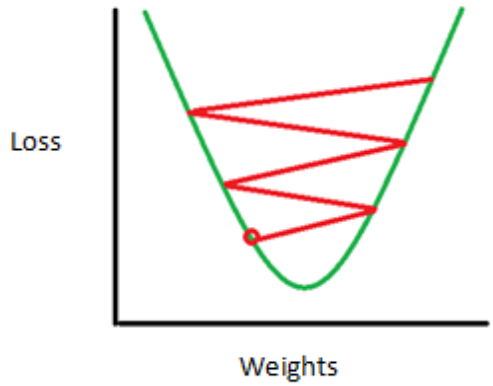
take all the weights of the filters and update them so that they change in the opposite direction of the gradient.

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

w = Weight
w_i = Initial Weight
η = Learning Rate

Eq.2 Stochastic Gradient Descend

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.



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

Fig. 2.8 Convergence of the function

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.

Overall we may summarize our network model with the following diagram

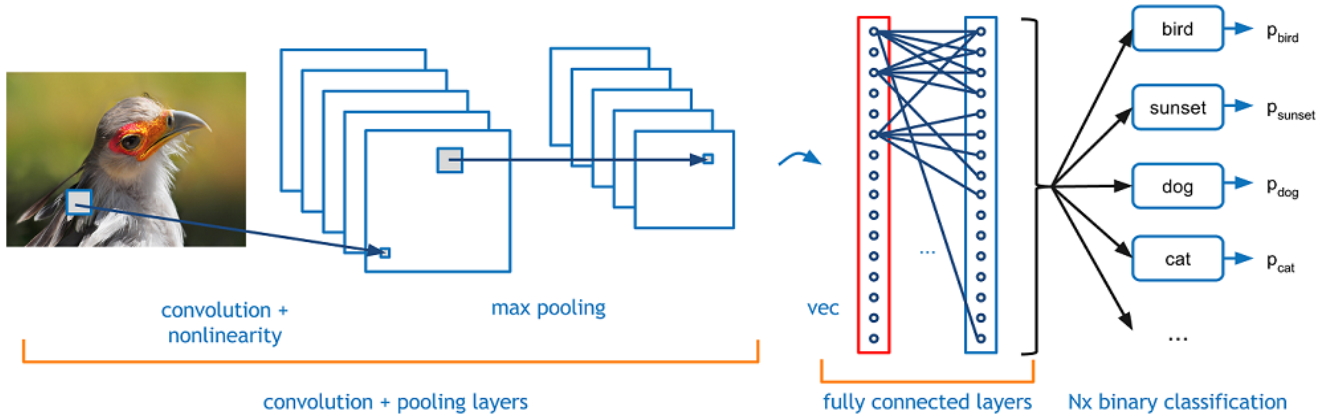


Fig. 2.9 Representation of the CNN model

3.4 Discriminator model:

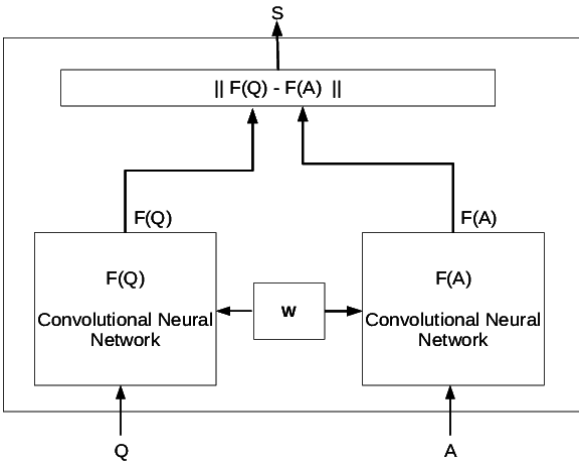


Fig. 2.10 Discriminator model

Deep-learning networks are distinguished from the more commonplace single-hidden-layer neural networks by their **depth**; that is, the number of node layers through which data passes in a multistep process of pattern recognition.

Earlier versions of neural networks such as the first were shallow, composed of one input and one output layer, and at most one hidden layer in between. More than three layers (including input and output) qualifies as “deep” learning. So deep is a strictly defined, technical term that means more than one hidden layer.

In deep-learning networks, each layer of nodes trains on a distinct set of features based on the previous layer's output. The further you advance into the neural net, the more complex the features your nodes can recognize, since they aggregate and recombine features from the previous layer.

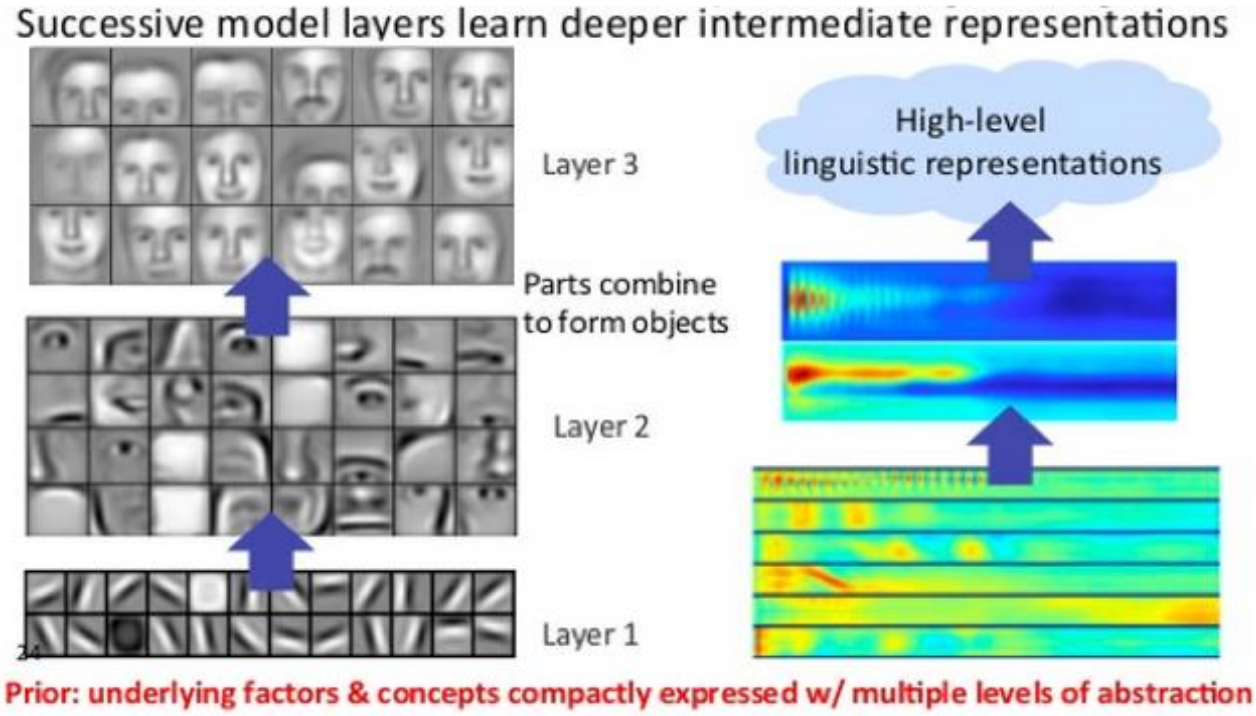


Fig 2.11 Training process in CNN

This is known as **feature hierarchy**, and it is a hierarchy of increasing complexity and abstraction. It makes deep-learning networks capable of handling very large, high-dimensional data sets with billions of parameters that pass through non-linear

Above all, these nets are capable of discovering latent structures within **unlabeled, unstructured data**, which is the vast majority of data in the world. Another word for unstructured data is *raw media*; i.e. pictures, texts, video and audio recordings. Therefore, one of the problems deep learning solves best is in processing and clustering the world's raw, unlabeled media, discerning similarities and anomalies in data that no human has organized in a relational database or ever put a name to.

For example, deep learning can take a million images, and cluster them according to their similarities: cats in one corner, ice breakers in another, and in a third all the photos of your grandmother. This is the basis of so-called smart photo albums.

Now apply that same idea to other data types: Deep learning might cluster raw text such as emails or news articles. Emails full of angry complaints might cluster in one corner of the vector space, while satisfied customers, or spambot messages, might cluster in others. This is the basis of various messaging filters, and can be used in customer-relationship management (CRM). The same applies to voice messages. With time series, data might cluster around normal/healthy behavior and anomalous/dangerous behavior. If the time series data is being generated by a smart phone, it will provide insight into users' health and habits; if it is being generated by an autopart, it might be used to prevent catastrophic breakdowns.

CHAPTER 4- SOFTWARE DESIGN

4. SOFTWARE DESIGN

The system has three fundamental tasks that it performs sequentially on all the individual data points. Hence they are made in the form of classes such that each data instance can be described as an object of a class. Apart from that a main module is added which acts as a pilot to all other mentioned class.

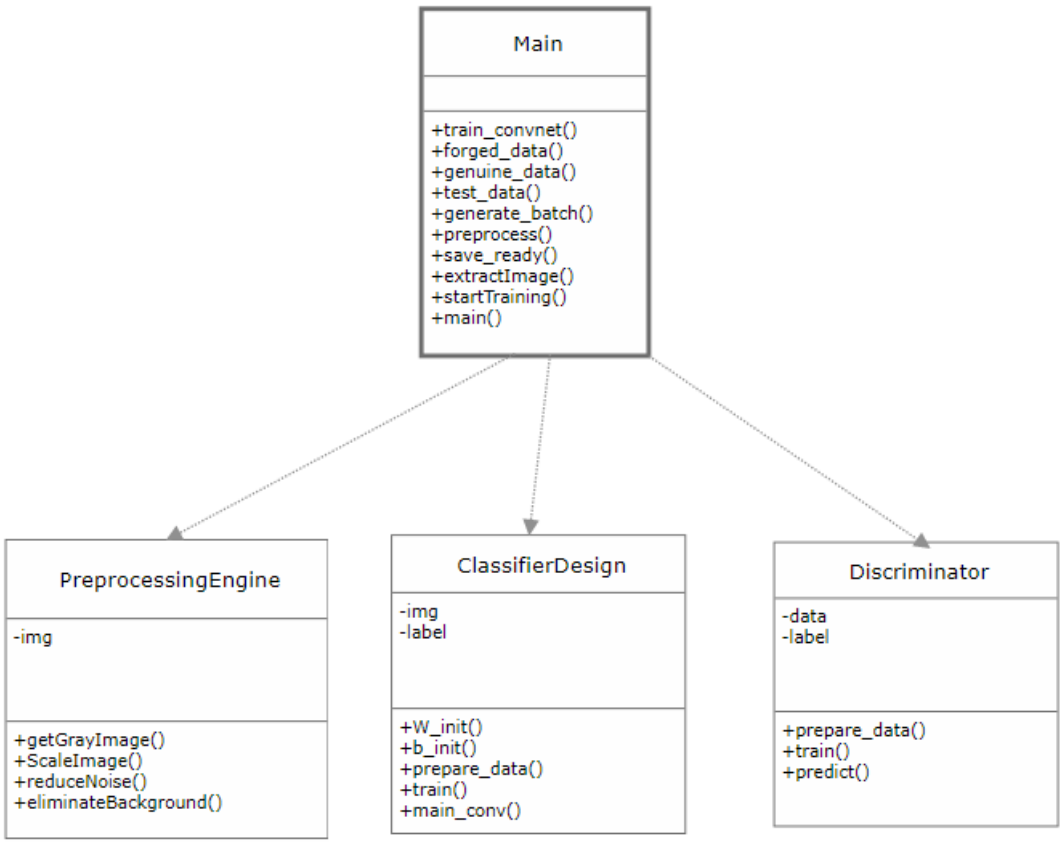


Fig 5.1 Class Diagram

The data flow diagram is shown next, which contains information of the process and its sub process. It's a 2 level DFD.

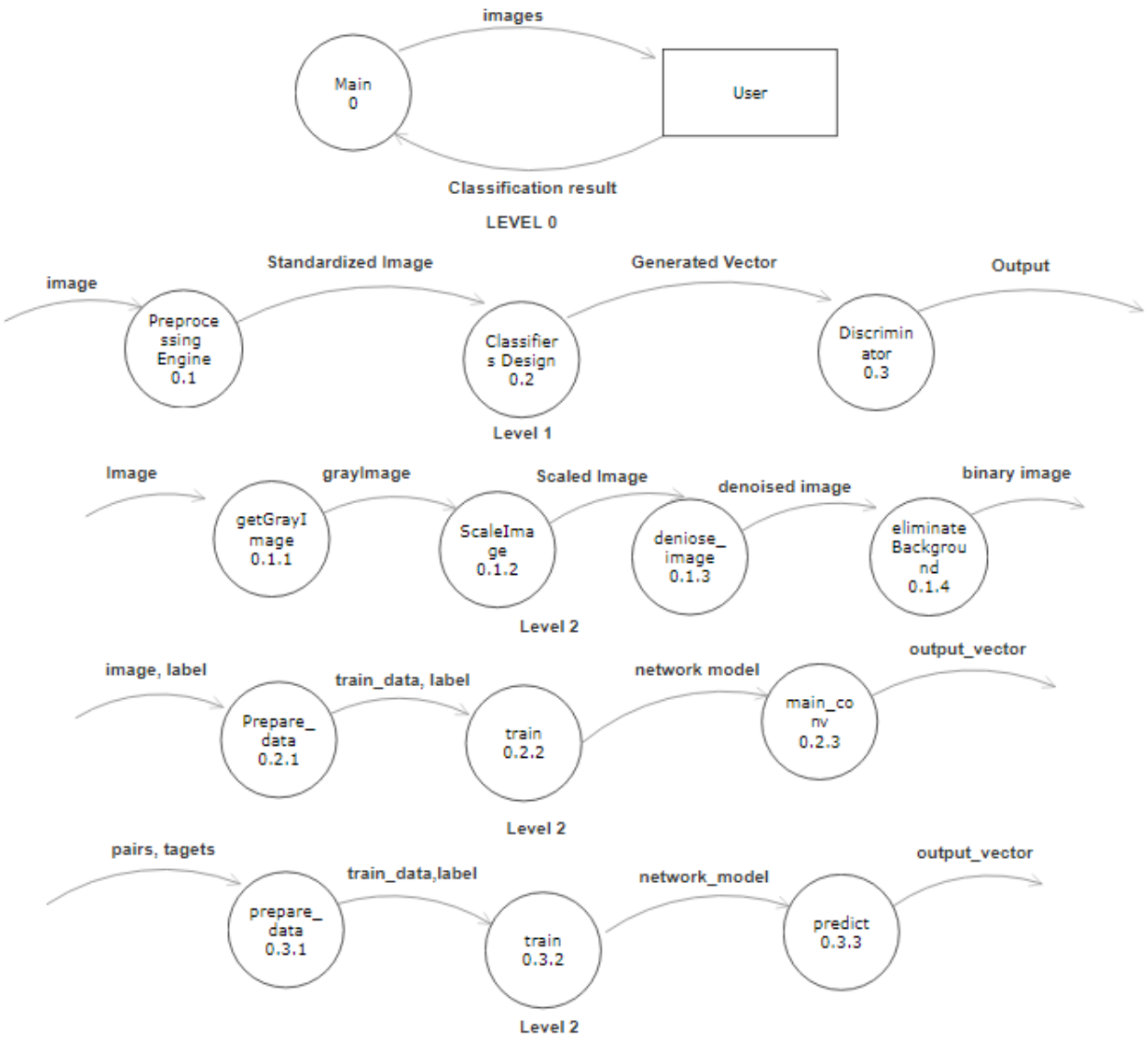


Fig 5.2 DFD Diagram

The network model is shown next. This model shows how the work flows across each module and visualizes the DFD and Class diagram

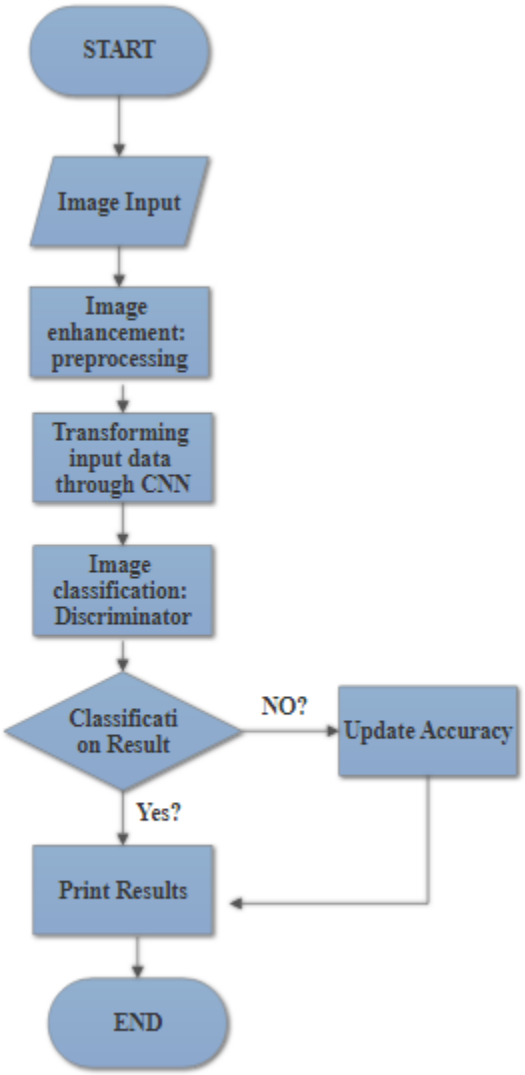


Fig 5.3 Flowchart

Chapter-5

SOFTWARE AND HARDWARE REQUIREMENTS

5. SOFTWARE AND HARDWARE REQUIREMENTS

HARDWARE REQUIREMENTS

1. Processor requirements (minimum 1.5GHz)
2. Memory requirements (Minimum 4GB RAM)

SOFTWARE REQUIREMENTS

1. Anaconda(32 bit windows installer package) distribution for Python 3.5 or greater
2. IPython console (comes with Anaconda installer package)
3. Numpy package compatible with python3.x (Library for Maths and Data analytics)
4. Scipy package compatible with python3.x (Machine Learning and Image processing library for python)

CHAPTER-6

CODE TEMPLATES

6 CODE TEMPLATES

Each class is associated with its own abstract class along with a factory class. The classes which are implemented here are PreprocessingEngine and ClassifierDesign. The template for which is described below.

6.1 CLASS PREPROCESSING ENGINE – It is used to carry out various Preprocessing Steps.

Table: -6.1

Constructor Summary	
PreprocessingEngine()	Numpy 2D Array object as parameter

Table: -6.2

Method Summary		
Parameters	Definition	Return Type
Numpy 2D Array	Converting to gray Image: getGrayImage()	Numpy 2D Array
Numpy 2D Array	Scaling Image to desired Dimensions: ScaleImage()	Numpy 2D Array
Numpy 2D Array	Denoising Image: reduceNoise()	Numpy 2D Array

Numpy 2D Array	Eliminate Background: eliminateBackground()	Numpy 2D Array

6.2 CLASS CLASSIFIER: This class is used as a classifier model. The contains the CNN model on which we are going to train our data. The summary of the CNN model is already given in the preceding section

Table: -6.3

Constructor Summary	
ClassifiersDesign	Image: Numpy 2D array Label: 1D array of shape (1,) indicating class of image Input_shape: indicating the shape of input(105,105,1) Batch_size: 1 Convent : A keras object

Table: -6.4

Method Summary		
Parameters	Definition	Return Type
Self, shape, name=None	W_init(); used for initialization and distribution of weights	Keras tensor
Self, shape, name=None	B_init();used for initialization and distribution of Biases	Keras Tensors

self	Prepare_data(); used for standardizing input data for training	Numpy array, numpy array
Self	Train(); used for training the network model	Keras.model object
self	Main_conv(); used for determining feature vector	Numpy array

6.3 CLASS DISCRIMINATOR- This class is used as a discriminator to understand and interpret the differences between outputs obtained from the CNN

Table: - 6.5

Constructor Summary	
Discriminator();	Self, data, label

Table: -6.6

Method Summary		
Parameters	Definition	Return Type
Self, data, label	Prepare_data(); used for standardizing input data();	Numpy array, numpy array
Self, data, label	Train(); used for training the model	Keras.model object
Self, data, label	Predict(); used for predicting the outputs	Numpy array

CHAPTER-7

TESTING

TESTING

1. PREPROCESSING MODULE

In preprocessing module we standardize the user input to use it for the process of learning or feeding into the module

We give inputs to the preprocessing class which are as follows

The inputs are images which may be of varying shapes and sizes

Some examples include

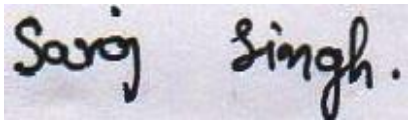


Fig.7.1 Signature specimen 1

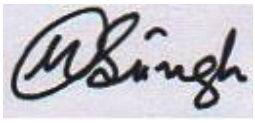


Fig.7.2 Signature specimen 2

After this we perform a series of operation on the image like gray scaling it, rescaling it, reducing noise and eliminating background

The results of preprocessed image are stored in the ready folder in the same package

Fig. 7.3 Signature specimen1 preprocessed

Fig.7.4. Signature specimen1 preprocessed

2. CONVOLUTIONAL NEURAL NETWORKS

As per our learning model each image needs to pass through a convolution neural network. The objective of this step is to transform our dataset into a form which is suitable form which can then be used for learning.

The utility of this is to first classify the exact class in which a signature belongs to and at the same time extract a sort of hash which tells us about the data. In other words our data which was a numpy array of shape (105,105) with values ranging from 0-255 will be transformed into an array of shape(10,) which will have class information along with signature information i.e a 2-tuple(class, hash)

Fig.7.5 Signature specimen 3

The above signature when fed into the network. Transforms it into a 2-tuple (class, hash).
The class is represented by its index and hash is represented by the value at index.
We can use this information to draw conclusions just by visualizing the output.
Here is an example
If we plot the array elements along with values of other images. Then the distribution may either superimpose upon each other, or may they may be different.

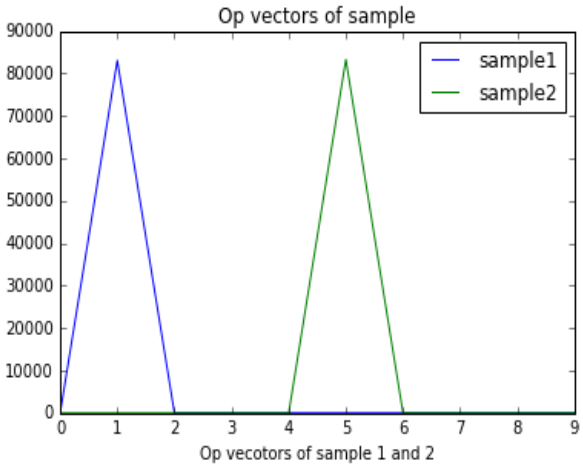


Fig. 7.6 Visualization of output for random forgery

Here is an example of a random forgery. As we can see that the real signature and the forged signature has been classified as belonging to different classes. Thus it is a case of random forgery

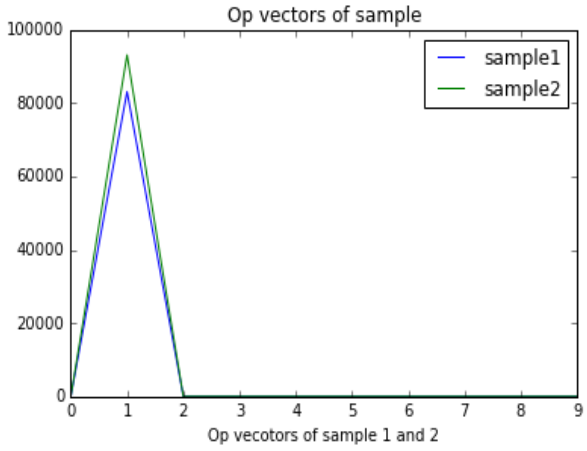


Fig. 7.7 Visualization of output for simple forgery

Here is an example of simple forgery. The signatures has been identified to belong to the same class. But the difference between their hash values is clearly visible. Thus, the signature sample is forged.

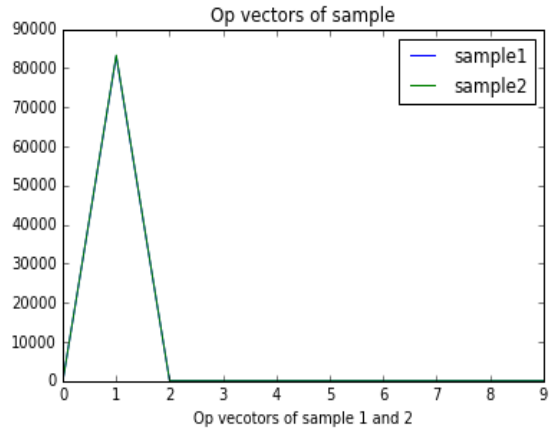
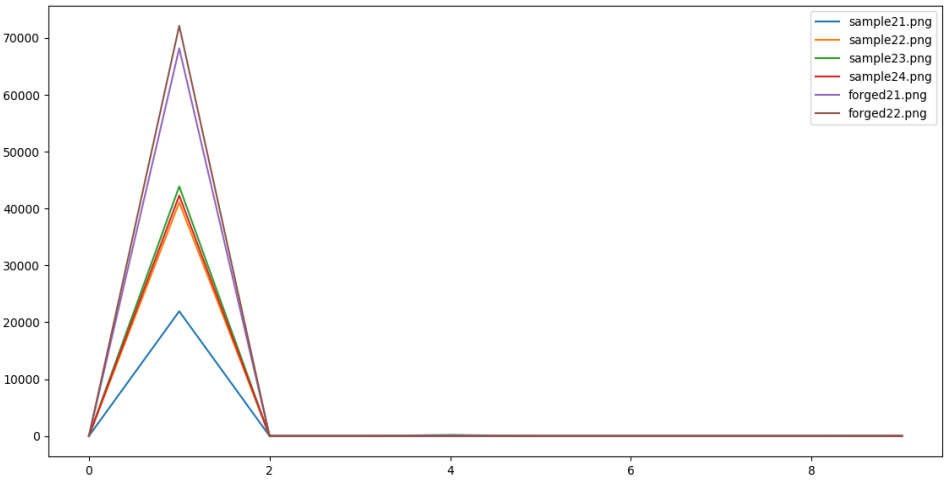


Fig. 7.8 Visualization of output for genuine signature

Here is another example of signature samples which are genuine. As we can see the graphs superimpose upon each other. Thus we can say that signatures appears genuine. But there can be a case of skilled forgery and for identifying that. We need to understand what differences are tolerable. Which we will be doing in the next section.

Also while testing CNN's we had a fight with the problem of overfitting. Sometimes the networks tend to over fit the data. To reduce overfitting. Dropouts were added to the network.

Upon testing on a dataset we can see that yellow, green and red lines are close together which suggests they are genuine signature samples. Violet and Brown lines shows large differences so they are correctly classified as forged. Al though the blue line is termed as genuine signature but the if we see the signature sample. The image was not taken properly, thus it shows deviations.



Op vector of samples

Fig. 7.9 model training summary

3. **Neural networks:** As we stated to learn which signature is genuine and which one is forged we have to understand how the differences can be interpreted

   The input to this network is a concatenated array of 20, 10 from each class of array received from the CNN

   The loss from the training is as follows

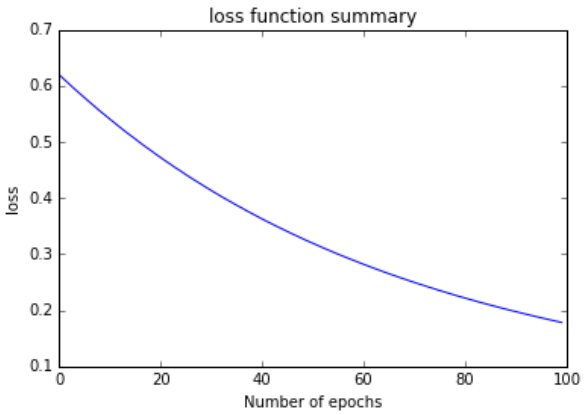


Fig. 7.10 Loss against epoch for discriminator

Chapter-8

CONCLUSION

6. CONCLUSION

Signature Identification and verification deals with the problem of identifying and verifying signature samples from a set of samples available to us. The task of static signature verification is a difficult vision problem within the field of biometrics because signature for an individual may change depending on the psychological factors of the person. Through this project, I am trying to develop a deep learning model for offline handwritten signatures recognition which is able to extract high-level representations.

Most of the models work well in the field only if the system is able to extract or create the right feature vector for a given Image. However the task is equally difficult. Thus we use a different kind of model in which we tend to extract high level representation of the model and thereby optimizing the feature vector required.

In our project we conclusively demonstrated how we can optimize feature vector and improve upon the accuracy of the overall task

Accurate Signature verification models have a wide range of applications ranging from banking to online transactions access control systems etc.

Chapter-9 FURTHER ENHANCEMENT

7. FURTHER ENHANCEMENT

From the point where we started the project last semester. We have come up with an entirely new set of model to solve our problem. The model works very well in classifying signatures. In the second part of the model we have used a Sequential non-recurrent neural network to learn from the transformed set of data. We can reduce this problem from a few shot learning to one shot learning if instead of the neural network we use. A differential function to classify signatures.

There are different forms of such distance function exists which finds out the difference between the arrays taking into account the distance between them. Using these distance functions the problem can be reduced to one shot learning

Apart from that I also feel the use of generalized adversarial network can also be useful and can further improve the performance of the system

Any further suggestions for improvement is welcomed and highly valued

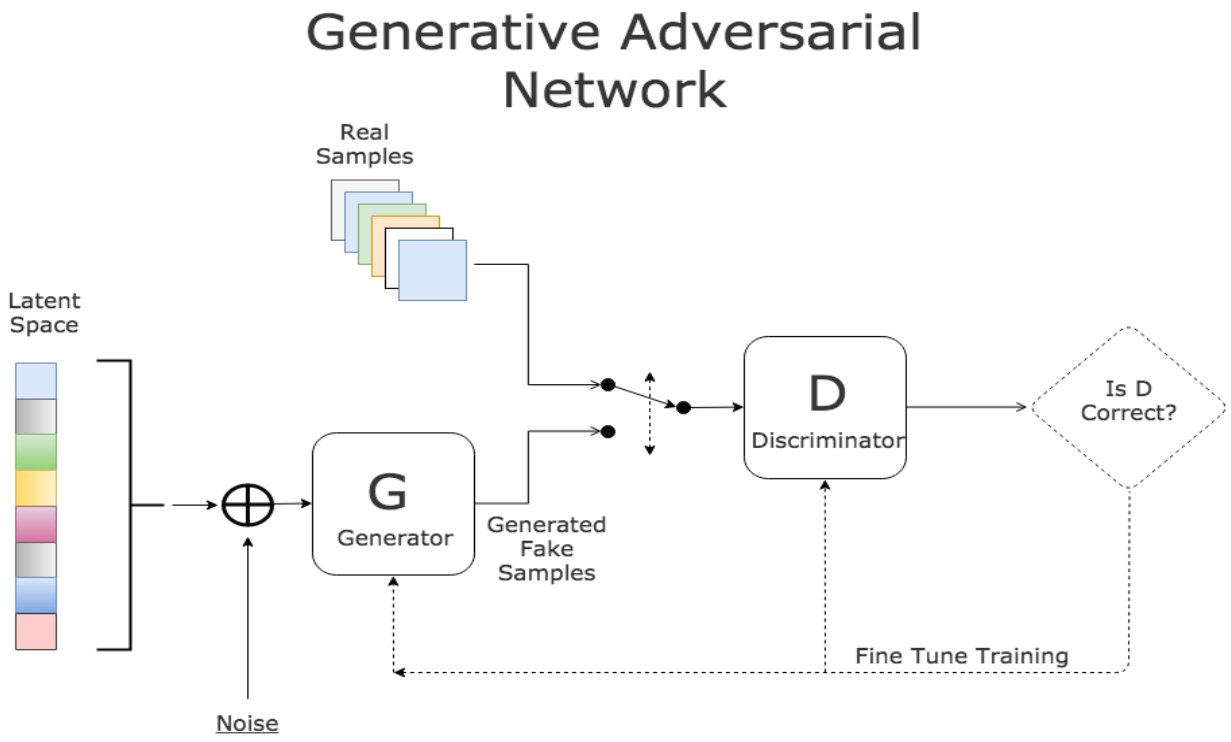


Fig 9.1 Generative Adversarial Network

APPENDICES

1. Rectified Linear Units

Instead of sigmoids, most recent deep learning networks use rectified linear units (ReLUs) for the hidden layers. A rectified linear unit has output 0 if the input is less than 0, and raw output otherwise. That is, if the input is greater than 0, the output is equal to the input. ReLUs' machinery is more like a real neuron in your body.

$$f(x) = max(x, 0)$$

ReLU activations are the simplest non-linear activation function you can use, obviously. When you get the input is positive, the derivative is just 1, so there isn't the squeezing effect you meet on backpropagated errors from the sigmoid function. Research has shown that ReLUs result in much faster training for large networks. Most frameworks like TensorFlow and TFLearn make it simple to use ReLUs on the the hidden layers, so you won't need to implement them yourself.

2. Softmax activation function:

The sigmoid function can be applied easily, the ReLUs will not vanish the effect during your training process. However, when you want to deal with classification problems, they cannot help much. Simply speaking, the sigmoid function can only handle two classes, which is not what we expect.

The softmax function squashes the outputs of each unit to be between 0 and 1, just like a sigmoid function. But it also divides each output such that the total sum of the outputs is equal to 1 (check it on the figure above).

The output of the softmax function is equivalent to a categorical probability distribution, it tells you the probability that any of the classes are true.

Mathematically the softmax function is shown below, where z is a vector of the inputs to the output layer (if you have 10 output units, then there are 10 elements in z). And again, j indexes the output units, so j = 1, 2, ..., K.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Eq. 3 softmax function

REFERENCE

[1] Alex Smola, SVN Vishwanathan, Introduction to Machine Learning, Department of Statistics and Computer Science, Purdue University.

[2] Gordan Zitkovic , Introduction to stochastic process- Lecture Notes 62(2), pp. 406-418

[3] Shiwani Sthapak, Minal Khopade, Chetana Kashid,, Artificial Neural Network based signature, recognition and verification, International Journal of Emerging Technology and Advanced. Engineering, August 2013.

[4] Harish Srinivasan, Sargur.N.Srihari, Matthew J Beal, Machine Learning for Signature Verification, Department of Computer Science and Engineering, State University of New York Center of excellence for document analysis and recognition(CEDAR).

[5] Mohsen Fayyaz, Mohommad Hajizadeh_saffar, Mohammad Sabokrou, Mahmood Fathy , Feature representation for online signature verification