A

Project Report On

Emotion Recognition

(Data Science Project)

Submitted in partial fulfilment for the award of

PG-DIPLOMA IN DEVELOPMENT OF BIG DATA ANALYTICS

Department of Big Data Analytics

Centre of Development of Advance Computing

Knowledge Park

Bengaluru

January 2017

Submitted To: -

Dr. Sarita

Submitted By:-

Rahul Basani

USN:-160850125008

Manisha Prasad Kalaga

USN:-160850125023

Sonali Gupta

USN:-160850125043

CERTIFICATE OF APPROVAL OF PROJECT WORK

This is to certify that the project report entitled "**Emotion Recognition**" is a bonafide work carried out by Rahul Basani (8), Manisha Prasad Kalaga (23) and Sonali Gupta (43) in fulfillment for the award of "**PG Diploma in Big Data Analytics"**

Place: CDAC-ACTS, Bangalore, Aug-2016 Batch.

Signature	Signature
(**Dr. Sarita**)	(**Ms. Janaki**)
Project Guide	Course Coordinator

ABSTRACT

Facial expressions play a significant role in human dialogue. As a result, there has been considerable work done on the recognition of facial expressions. The application of our research will be beneficial in improving human-machine dialogue. One can imagine the improvements to computer interfaces, automated clinical (psychological) research or even interactions between humans and autonomous robots.

In this project we tried to develop our own facial expression detection model based on machine learning, computer vision and big data analytics tools. The system involves pre-processing image data by normalizing and rotating the image, and then feeding this to a multi-layer convolutional neural network for classification and recognition of expressions. We used python for programming and Apache Spark as execution engine to process large scale data.

Contents

INTRODUCTION

Researchers have found 80% of the communication in any conversation happens through the non verbal means and facial gestures or expressions. We human say a lot through our expressions, facial gestures are insight to a person's internal emotional state, intentions or social communication. This gave way for facial expressions to be used as new information and control source for computers.

Facial expression analysis has been an active research topic for behavioral scientists since the work of Darwin in 1872 presented an early attempt to automatically analyze facial expressions by tracking the motion of 20 identified spots on an image sequence in 1978. After that, much progress has been made to build computer systems to help us understand and use this natural form of human communication. The field of facial recognition opened up a big space of opportunity for a large variety of applications such as data-driven animation, neuro-marketing, interactive games, sociable robotics and many other human-computer interaction systems.

Several facial expression recognition approaches were developed in the last decades with an increasing progress in recognition performance. An important part of this recent progress was achieved thanks to the emergence of Deep Learning methods and more specifically with Convolutional Neural Networks, which is one of the deep learning approaches. These approaches became feasible due to: the larger amount of data available nowadays to train learning methods and the advances in GPU technology.

PROJECT OVERVIEW

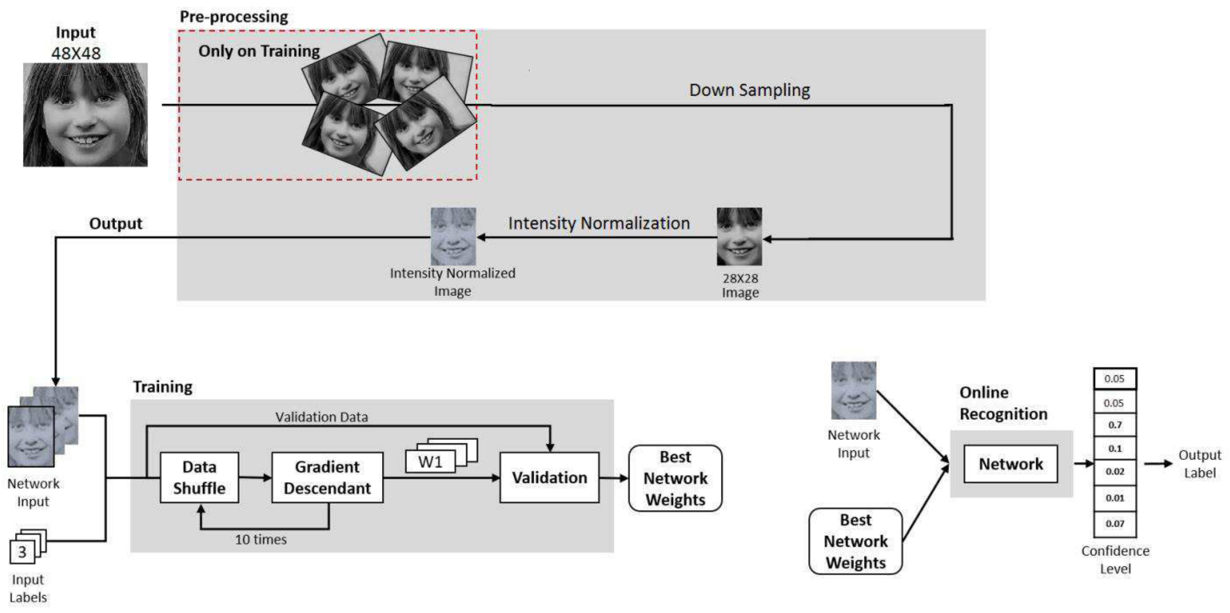
The goal of this project is to predict, from the gray scale picture of a person's face, which emotion the facial expression conveys. We will be predicting seven emotions namely anger, disgust, fear, happiness, sadness, surprise and neutral. Our evaluation metric will be the accuracy for each emotion (fraction of correctly classified images), supplemented by a confusion matrix which highlights which emotions are better recognized than others.

Input: 48 by 48 gray scale image of a face
Output: Emotion conveyed by facial expression

PROJECT DESCRIPTION

Methodology

The system operates in two main phases: training and test. During training, the system receives a training data comprising gray scale images of faces learns a set of weights for the network. To ensure that the training performance is not aff ected by the order of presentation of the examples, a few images are separated as validation and are used to choose the final best set of weights out of a set of trainings performed with samples presented in diff erent orders. During test, the system receives a grayscale image of a face and outputs the predicted expression by using the final network weights learned during training. We chose Tensor Flow as our framework to train neural networks on. Tensor Flow is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.



Data Augmentation

To compensate for the relatively small size of the dataset, we made use of the popular data augmentation technique that consists in flipping images horizontally and rotating by an angle of .
As emotions should intuitively not change based on whether or not facial expressions are mirrored, it seemed a sensible choice.

Down-sampling

The down-sampling operation is performed to reduce the image size for the network and to ensure scale normalization, i.e. the same location for the face components (eyes, mouth, eyebrow, etc) in all images. The downsampling uses a linear interpolation approach. This procedure helps the CNN to learn which regions are related to each specific expression. The final image is down-sampled, using a area interpolation, to 28×28 pixels.

Intensity Normalization

The image brightness and contrast can vary even in images of the same person in the same expression, therefore, increasing the variation in the feature vector. Such variations increase the complexity of the problem that the classifier has to solve for each expression. In order to reduce these issues an intensity normalization was applied. A method adapted from a bio-inspired technique called contrastive equalization, was used. Basically, the normalization is a two step procedure: firstly a subtractive local contrast normalization is performed; and secondly, a divisive local contrast normalization is applied. Equation 1 shows how each new pixel value is calculated in the intensity normalization procedure:

$$x0 = (x - 128.0)/128.0 \qquad (1)$$

where, x0 is the new pixel value, x is the original pixel value.

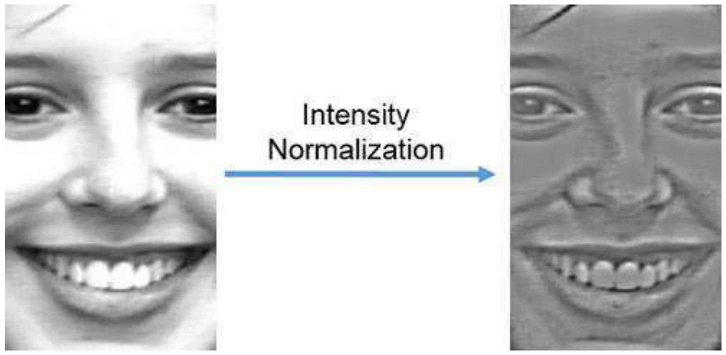


Fig: Intensity Normalized Image

Architecture

The final architecture retained can be described as follows:
- 5 X 5 Conv - ReLU – 2 X 2 Max-Pool with 16 filters
- 5 X 5 Conv - ReLU – 2 X 2 Max-Pool with 16 filters
- FC hidden layer with 3136 hidden units
- FC layer to 7 class scores

LITERATURE SURVEY

Machine Learning

Machine learning is a type of artificial intelligence (AI) that provides computers with the ability to learn without being explicitly programmed. Machine learning focuses on the development of computer programs that can teach themselves to grow and change when exposed to new data.

The process of machine learning is similar to that of data mining. Both systems search through data to look for patterns. However, instead of extracting data for human comprehension -- as is the case in data mining applications – machine learning uses that data to detect patterns in data and adjust program actions accordingly. Machine learning algorithms are often categorized as being supervised or unsupervised. Supervised algorithms can apply what has been learned in the past to new data. Unsupervised algorithms can draw inferences from datasets. Here are several examples:

• **Optical character recognition:** categorize images of handwritten characters by the letters represented.

• **Face detection:** find faces in images (or indicate if a face is present).

• **Spam filtering:** identify email messages as spam or non-spam.

• **Topic spotting:** categorize news articles (say) as to whether they are about politics, sports, entertainment, etc.

• **Spoken language understanding:** within the context of a limited domain, determine the meaning of something uttered by a speaker to the extent that it can be classified into one of a fixed set of categories.

Supervised Learning

Supervised learning is the machine learning task of inferring a function from labeled training data. The training data consist of a set of training examples. In supervised learning, each example is a pair consisting of an input object (typically

a vector) and a desired output value (also called the supervisory signal). A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new examples. An optimal scenario will allow for the algorithm to correctly determine the class labels for unseen instances. Speech recognition using hidden Markov models and Bayesian networks relies on some elements of supervision as well in order to adjust parameters to, as usual, minimize the error on the given inputs.
Notice something important here: in the classification problem, the goal of the learning algorithm is to minimize the error with respect to the given inputs. These inputs, often called the "training set", are the examples from which the agent tries to learn. But learning the training set well is not necessarily the best thing to do. For instance, if I tried to teach you exclusive-or, but only showed you combinations consisting of one true and one false, but never both false or both true, you might learn the rule that the answer is always true. Similarly, with machine learning algorithms, a common problem is over-fitting the data and essentially memorizing the training set rather than learning a more general classification technique.

Unsupervised Learning

Unsupervised learning is the machine learning task of inferring a function to describe hidden structure from unlabeled data. Since the examples given to the learner are unlabeled, there is no error or reward signal to evaluate a potential solution. This distinguishes unsupervised learning from supervised learning and reinforcement learning.
Unsupervised learning is closely related to the problem of density estimation in statistics. However unsupervised learning also encompasses many other techniques that seek to summarize and explain key features of the data.

Neural Network Architectures

Layer-Wise Organization

Neural Networks are modeled as collections of neurons that are connected in an acyclic graph. In other words, the outputs of some neurons can become inputs to other neurons. Cycles are not allowed since that would imply an infinite loop in the forward pass of a network. Instead of an amorphous blobs of connected neurons, Neural Network models are often organized into distinct layers of neurons. For regular neural networks, the most common layer type is the **fully-connected layer** in which neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections. Below are two example Neural Network topologies that use a stack of fully-connected layers:

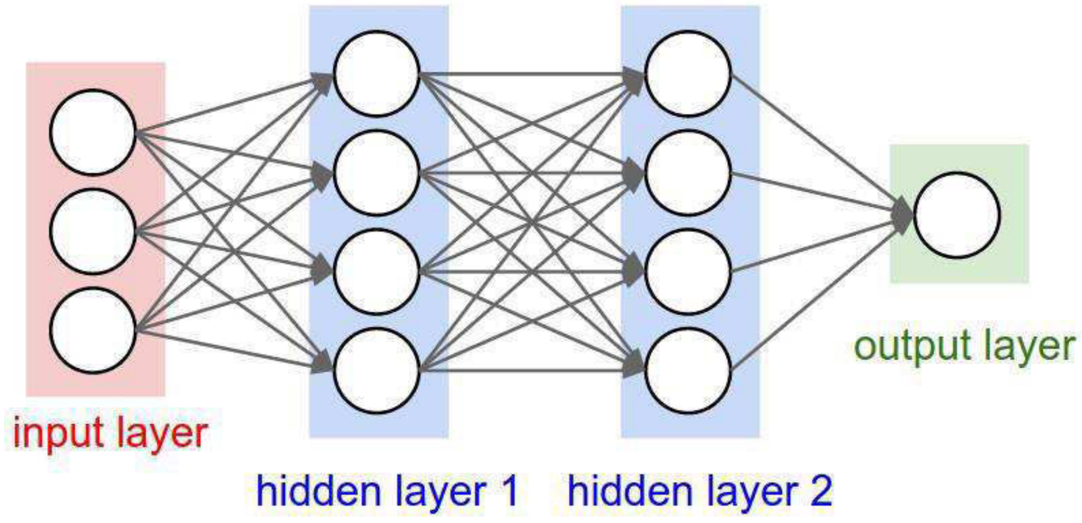
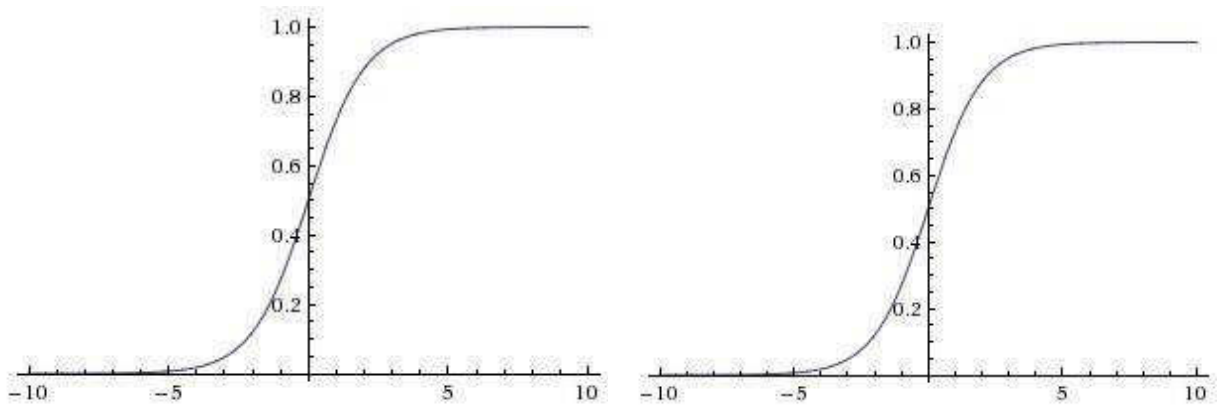


Fig: A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Notice that in both cases there are connections (synapses) between neurons across layers, but not within a layer.

Activation Function

Every activation function (or *non-linearity*) takes a single number and performs a certain fixed mathematical operation on it. There are several activation functions you may encounter in practice:



Left: Sigmoid non-linearity squashes real numbers to range between [0,1] **Right:** The tanh non-linearity squashes real numbers to range between [-1,1].

Sigmoid. The sigmoid non-linearity has the mathematical form σ(x)=1/(1+e−x)σ(x)=1/(1+e−x) and is shown in the image above on the left. As alluded to in the previous section, it takes a real-valued number and "squashes" it into range between 0 and 1. In particular, large negative numbers become 0 and large positive numbers become 1. The sigmoid function has seen frequent use historically since it has a nice interpretation as the firing rate of a neuron: from not firing at all (0) to fully-saturated firing at an assumed maximum frequency (1). In practice, the sigmoid non-linearity has recently fallen out of favor and it is rarely ever used.

Tanh. The tanh non-linearity is shown on the image above on the right. It squashes a real-valued number to the range [-1, 1]. Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centered. Therefore, in practice the *tanh non-linearity is always preferred to the sigmoid nonlinearity.* Also note that the tanh neuron is simply a scaled sigmoid neuron, in particular the following holds: $\tanh(x)=2\sigma(2x)-1\tanh(x)=2\sigma(2x)-1$.

ReLU. The Rectified Linear Unit has become very popular in the last few years. It computes the function $f(x)=\max(0,x)f(x)=\max(0,x)$. In other words, the activation is simply thresholded at zero (see image above on the left). There are several pros and cons to using the ReLUs:
∟ (+) It was found to greatly accelerate (e.g. a factor of 6 in Krizhevsky et al.) the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form. (+) Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero.
(-) Unfortunately, ReLU units can be fragile during training and can "die". For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training since they can get knocked off the data manifold. For example, you may find that as much as 40% of your network can be "dead" (i.e. neurons that never activate across the entire training dataset) if the learning rate is set too high. With a proper setting of the learning rate this is less frequently an issue.

Convolutional Neural Network

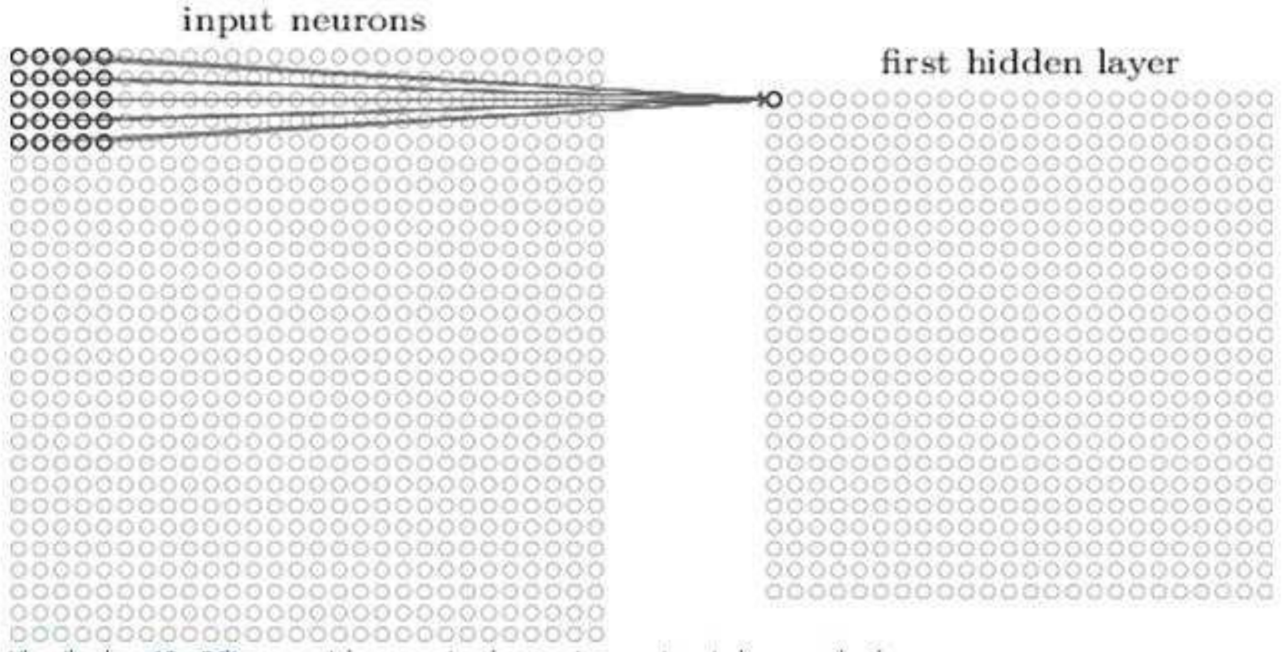
In machine learning, a **convolutional neural network** (**CNN**, or **ConvNet**) is a type of feed-forward artificial neural network in which the connectivity pattern between its neurons is inspired by the organization of the animal visual cortex. Individual cortical neurons respond to stimuli in a restricted region of space known as the receptive field. The receptive fields of different neurons partially overlap such that they tile the visual field. The response of an individual neuron to stimuli within its receptive field can be approximated mathematically by a convolution operation. Convolutional networks were inspired by biological processes and are variations of multilayer perceptrons designed to use minimal amounts of preprocessing. They have wide applications in image and video recognition, recommender systems and natural language processing.

Convolutional neural networks (CNNs) consist of multiple layers of receptive fields. These are small neuron collections which process portions of the input image. The outputs of these collections are then tiled so that their input regions overlap, to obtain a better representation of the original image; this is repeated for every such layer. Tiling allows CNNs to tolerate translation of the input image.

Convolutional networks may include local or global pooling layers, which combine the outputs of neuron clusters. They also consist of various combinations of convolutional and fully connected layers, with pointwise nonlinearity applied at the end of or after each layer. A convolution operation on small regions of input is introduced to reduce the number of free parameters and improve generalization. One major advantage of convolutional networks is the use of shared weight in convolutional layers, which means that the same filter (weights bank) is used for each pixel in the layer; this both reduces memory footprint and improves performance.

Convolutional Layer

The first layer in a CNN is always a Convolutional 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 dimensions of this filter is 5 x 5 x 3. Now, let's take the first position the filter is in for example. It would be the top left corner. As the filter is sliding, or convolving, around the input image, it is multiplying the values in the filter with the original pixel values of the image (aka computing element wise multiplications). These multiplications are all summed up (mathematically speaking, this would be 75 multiplications in total). So now 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.



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

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

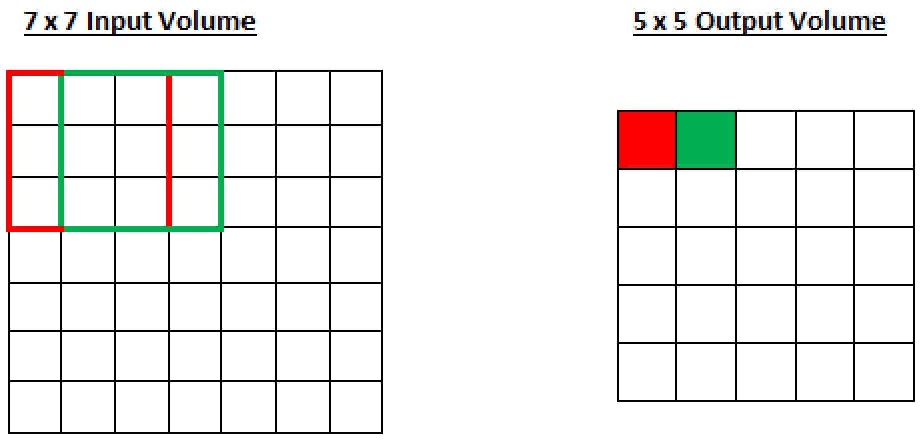
Fully Connected Layer

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

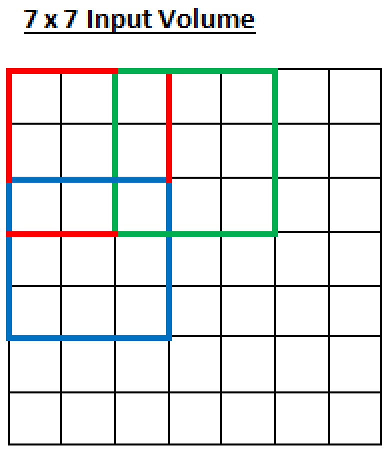
Stride and Padding

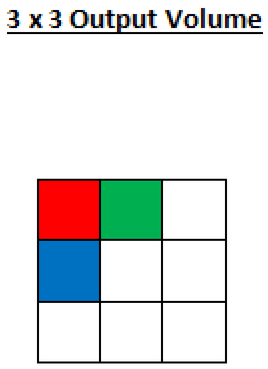
Alright, let's look back at our good old conv layers. Remember the filters, the receptive fields, the convolving? Good. Now, there are 2 main parameters that we can change to modify the behavior of each layer. After we choose the filter size, we also have to choose the **stride** and the **padding**.

Stride controls how the filter convolves around the input volume. In the example we had in part 1, the filter convolves around the input volume by shifting one unit at a time. The amount by which the filter shifts is the stride. In that case, the stride was implicitly set at 1. Stride is normally set in a way so that the output volume is an integer and not a fraction. Let's look at an example. Let's imagine a 7 x 7 input volume, a 3 x 3 filter (Disregard the 3rd dimension for simplicity), and a stride of 1. This is the case that we're accustomed to.



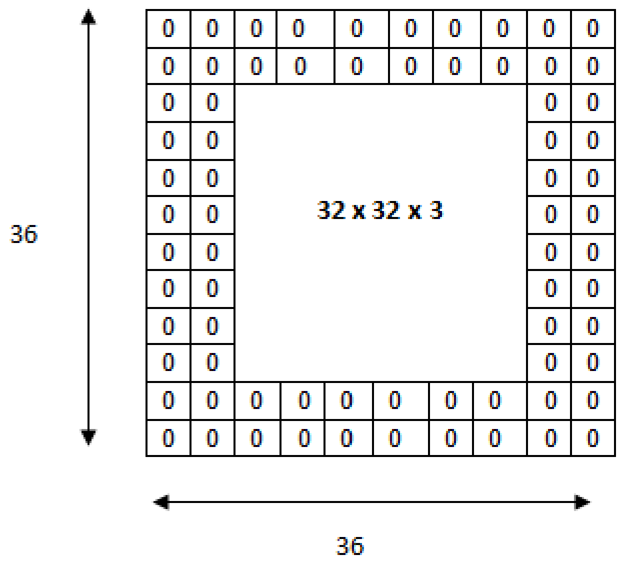
Same old, same old, right? See if you can try to guess what will happen to the output volume as the stride increases to 2.





So, as you can see, the receptive field is shifting by 2 units now and the output volume shrinks as well. Notice that if we tried to set our stride to 3, then we'd have issues with spacing and making sure the receptive fields fit on the input volume. Normally, programmers will increase the stride if they want receptive fields to overlap less and if they want smaller spatial dimensions.

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



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

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

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

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

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

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

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

Choosing Hyperparameters

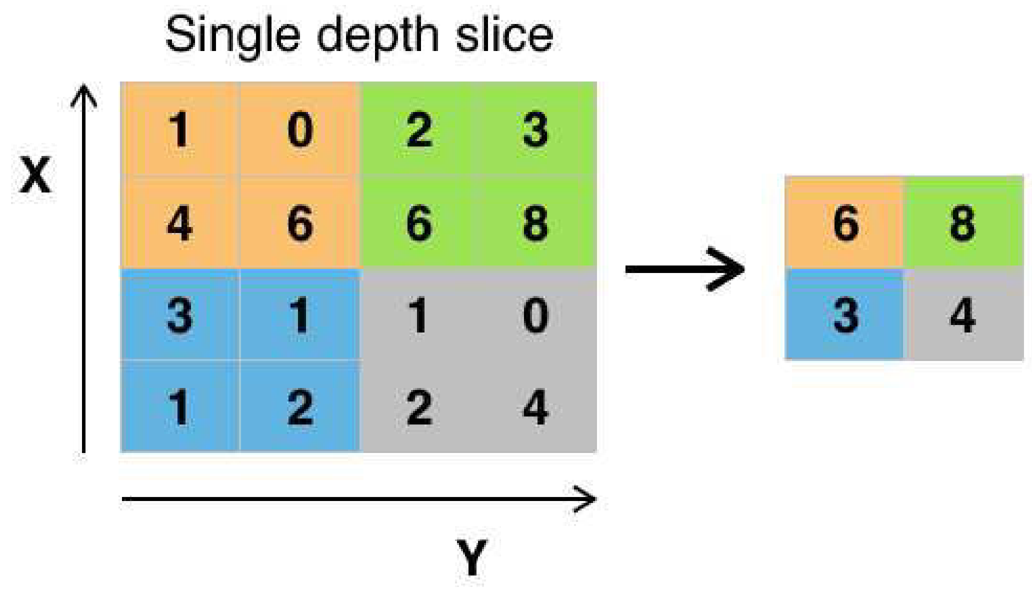
How do we know how many layers to use, how many conv layers, what are the filter sizes, or the values for stride and padding? These are not trivial questions and there isn’t a set standard that is used by all researchers. This is because the network will largely depend on the type of data that you have.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

ReLU (Rectified Linear Units) Layers

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

Pooling Layers

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



Example of Maxpool with a 2x2 filter and a stride of 2

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

Dropout Layers

Now, **dropout layers** have a very specific function in neural networks. In the last section, we discussed the problem of overfitting, where after training, the weights of the network are so tuned to the training examples they are given that the network doesn't perform well when given new examples. The idea of dropout is simplistic in nature. This layer "drops out" a random set of activations in that layer by setting them to zero in the forward pass. Simple as that. Now, what are the benefits of such a simple and seemingly unnecessary and counterintuitive process? Well, in a way, it forces the network to be redundant. By that I mean the network should be able to provide the right classification or output for a specific example even if some of the activations are dropped out. It makes sure that the network isn't getting too "fitted" to the training data and thus helps alleviate the overfitting problem. An important note is that this layer is only used during training, and not during test time.

Training

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.

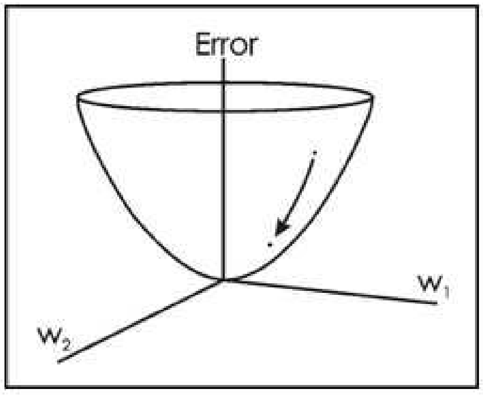
Before we get into backpropagation, we must first take a step back and talk about what a neural network needs in order to work. At the moment we all were born, our minds were fresh. We didn't know what a cat or dog or bird was. In a similar sort of way, before the CNN starts, the weights or filter values 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.

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

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.



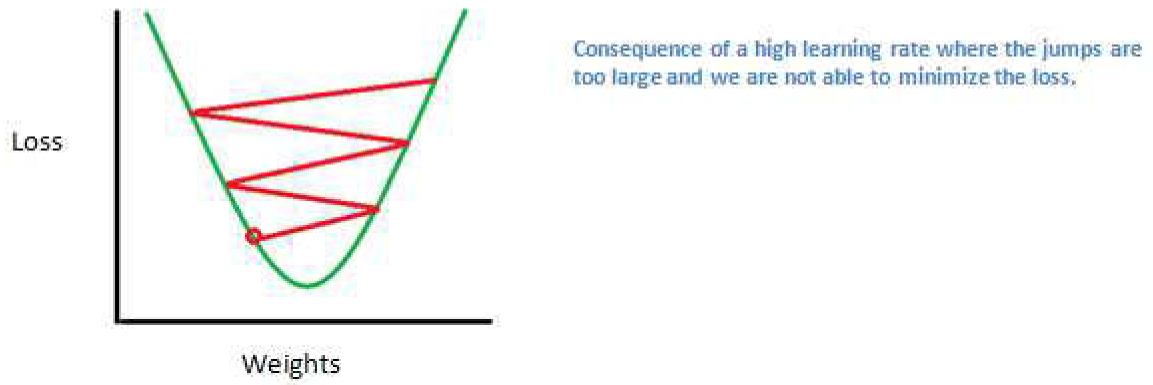
One way of visualizing this idea of minimizing the loss is to consider a 3-D graph where the weights of the neural net (there are obviously more than 2 weights, but let's go for simplicity) are the independent variables and the dependent variable is the loss. The task of minimizing the loss involves 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.

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 direction of the gradient.

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

w = Weight
w_i = Initial Weight
η = Learning Rate

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


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

The process of forward pass, loss function, backward pass, and parameter update is generally called one epoch. The program will repeat this process for a fixed number of epochs for each training image. 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.

LIBRARIES

Python Libraries

NumPy: NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object.
- sophisticated (broadcasting) functions.
- tools for integrating C/C++ and Fortran code.
- useful linear algebra, Fourier transform, and random number capabilities.

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

OS : This module provides a portable way of using operating system dependent functionality. If you just want to read or write a file see open(), if you want to manipulate paths, see the os.path module, and if you want to read all the lines in all the files on the command line see the fileinput module. For creating temporary files and directories see the tempfile module, and for high-level file and directory handling see the shutil module.

OpenCV : An important feature of Python is that it can be easily extended with C/C++. This feature helps us to write computationally intensive codes in C/C++ and create a Python wrapper for it so that we can use these wrappers as Python modules. This gives us two advantages: first, our code is as fast as original C/C++ code (since it is the actual C++ code working in background) and second, it is very easy to code in Python. This is how OpenCV-Python works, it is a Python wrapper around original C++ implementation. And the support of Numpy makes the task more easier.

Pandas : Pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series. pandas is free software released under the three-clause BSD license. The name is derived from the term "Panel data", an econometrics term for multidimensional structured data sets.

TensorFlow : It is an open source software library for machine learning in various kinds of perceptual and language understanding tasks. It is currently used for both research and production by 50different teams in dozens of commercial Google products, such as speech recognition, Gmail, Google Photos, and search, many of which had previously used its predecessor DistBelief. TensorFlow was originally developed by the Google Brain team for Google's research and production purposes and later released under the Apache 2.0 open source license on November 9, 2015. Among the broad spectrum of applications TensorFlow is the foundation for, it has been successfully implemented in automated image captioning software, such as DeepDream .Google officially implemented RankBrain on 26 October 2015, backed by TensorFlow. RankBrain now handles a substantial number of search queries, replacing and supplementing traditional static algorithm based search results.

SYSTEM ENVIRONMENT

Operating Systems: Windows 10 and Linux (Ubuntu 16.04 LTS)

Hardware: Intel® Core™ i5-5200U CPU @ 2.20GHz
RAM 8GB
HDD 250GB

Software: Anaconda 3-4.2.0 Linux-x86_64

Technologies used:
Spark 2.0.1
Python 3.5.2

Libraries for Python:
Numpy
OpenCV 3.1.0
OS
Pandas
Tensor Flow 0.12.0rc0

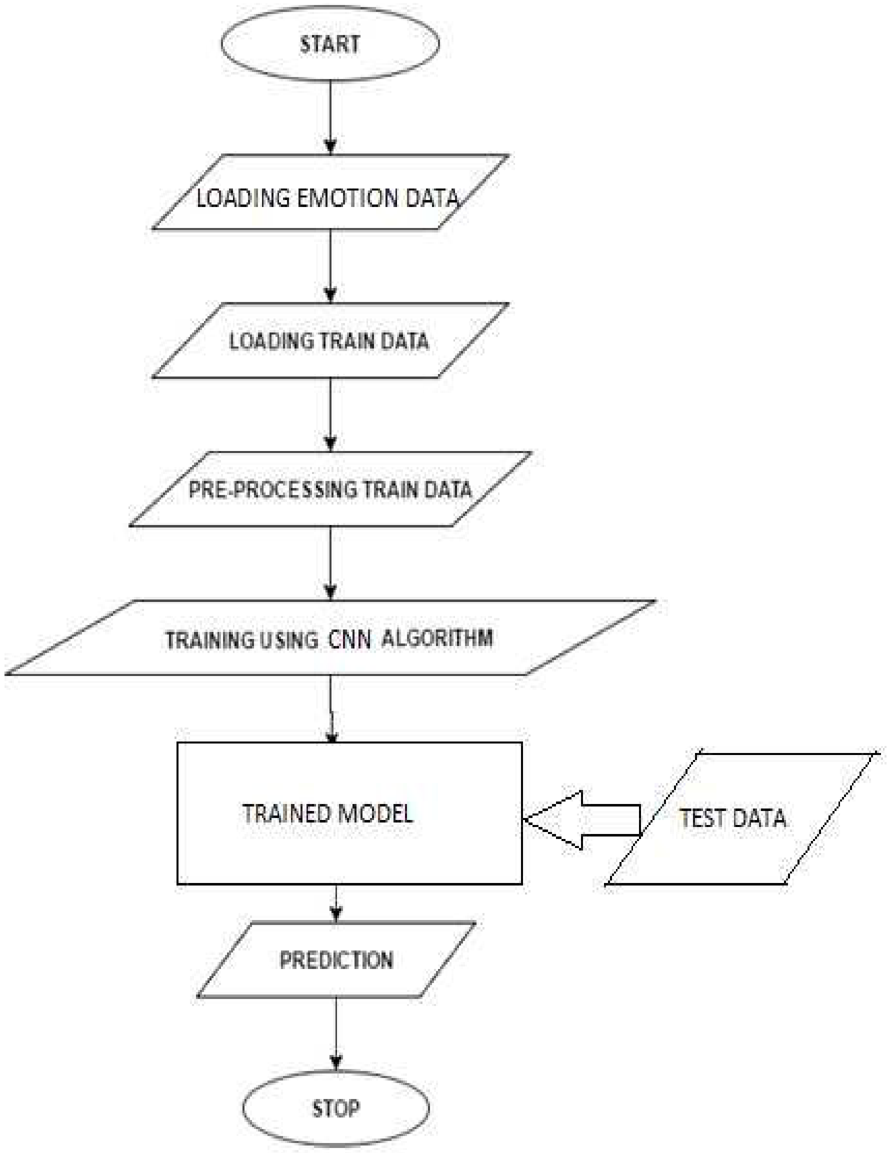
DATA SET

The data we used comprises of 48 X 48 pixel gray scale images of faces from the Kaggle competition Challenges in Representation Learning: Facial Expression Recognition Challenge . The training set consists of 28,709 examples, while both the test and validation sets are composed of 3,589 examples. This data set contains photos and labels of seven categories of facial expressions/emotions: Anger, Disgust, Fear, Happy, Sad, Surprise, Neutral. These images are mostly centered and adjusted so that the face occupies about the same amount of space in each image.

PROGRAM EXECUTION FLOW

- Loading of train data.
- Preprocessing of train data i.e resizing, rotating and normalizing.
- Training using Convolutional Neural network algorithm.
- Loading of test data .
- Preprocessing of test data.
- Applying trained model for prediction.

Flow Diagram



SOURCE CODE

```
# coding: utf-8



from pyspark import SparkConf,SparkContext

#creating spark context
con = SparkConf().setAppName("loading emotion
csv").setMaster("local[4]")
sc = SparkContext(conf = con)

#initializing start time
start_time = time.time()

#reading emotion data csv file
def get_emotion_data(filename):
   rdd = sc.textFile(filename)
   emotions = rdd.map(lambda line: line.split(',')[0]).collect()
   pixels = rdd.map(lambda line: line.split(',')[1]).collect()
   data = pd.DataFrame({"emotion":emotions[1:],"pixels":pixels[1:]})
   return data

#resizing emotion data
def data_resize(filename):
   data = get_emotion_data("filePath")
   emotions , data_pixels = data['emotion'] , data['pixels']
   total_length = len(emotions)
   data_pixel_df = []
   emotion_list = []
   image_list = []
   for i in range(total_length):

      for line in data_pixels[i].split(" "):
         data_pixel_df.append(float(line))
```

```
        data_pixel_df_array = np.array(data_pixel_df)

        data_pixel_df_array = data_pixel_df_array.reshape(48,48)
        resize    =    cv2.resize(data_pixel_df_array,(28,28),interpolation    =
cv2.INTER_AREA)

        #ab is resized image
        ab = resize.reshape(784,)
        resize_image_in_str = " ".join(str(int(a)) for a in ab)
        emotion_list.append(emotions[i])
        image_list.append(resize_image_in_str)
        data_pixel_df=[]
    resized_dataframe = pd.DataFrame({
                    'emotion': emotion_list,
                    'pixels' : image_list
                    })
    return resized_dataframe

resized_dataset = data_resize("fer2013_final.csv")




dataSet = resized_dataset

#separating dataframe as per emotions
emotion_0 = dataSet[dataSet['emotion']==0]
emotion_1 = dataSet[dataSet['emotion']==1]
emotion_2 = dataSet[dataSet['emotion']==2]
emotion_3 = dataSet[dataSet['emotion']==3]
emotion_4 = dataSet[dataSet['emotion']==4]
emotion_5 = dataSet[dataSet['emotion']==5]
emotion_6 = dataSet[dataSet['emotion']==6]

del dataSet #to free memory

#to separate train and test dataset
def separate_train_and_test(df,fraction=0.2):
    #to shuffle dataset
    a = df.sample(frac=1).reset_index(drop=True)
    length = int(a.shape[0]*fraction)
    return a[:length], a[length+1:]

#separated dataset as per emotions
emotion_0_test , emotion_0_train = takeRandom(emotion_0)
emotion_1_test , emotion_1_train = takeRandom(emotion_1)
emotion_2_test , emotion_2_train = takeRandom(emotion_2)
emotion_3_test , emotion_3_train = takeRandom(emotion_3)
emotion_4_test , emotion_4_train = takeRandom(emotion_4)
emotion_5_test , emotion_5_train = takeRandom(emotion_5)
emotion_6_test , emotion_6_train = takeRandom(emotion_6)

#test dataset
testDataSet                                                              =
pd.concat([emotion_0_test,emotion_1_test,emotion_2_test,emotion_3_test,
emotion_4_test,emotion_5_test,emotion_6_test])

#train dataset
trainDataSet                                                             =
pd.concat([emotion_0_train,emotion_1_train,emotion_2_train,emotion_3_tr
ain,emotion_4_train,emotion_5_train,emotion_6_train])

#writing datasets to csv
testDataSet.to_csv('testDataset.csv',index=False)
trainDataSet.to_csv('trainDataset.csv',index=False)



#data augmentation by mirroring and rotating
def data_augment(filename):

    data = pd.read_csv(filename)
    emotions , data_pixels = data['emotion'] , data['pixels']
    total_length = len(emotions)
    data_pixel_df = np.array([])
    emotion_list = np.array([])
    image_list = np.array([])
    for i in range(total_length):
        emotion_list = np.append(emotions[i])

        for x in data_pixels[i].split(" "):
            data_pixel_df = np.append(data_pixel_df,x)


        data_pixel_df_array = np.array(data_pixel_df,dtype = 'uint8')
        original_image = " ".join(str(a) for a in data_pixel_df_array)
        image_list = np.append(image_list,original_image)

        #flip image
        data_pixel_df_array = data_pixel_df_array.reshape(48,48)
        flipped_image = cv2.flip(data_pixel_df_array,1)

        #rotate image
        image_center = tuple(np.array(data_pixel_df_array.shape)/2)
        rotation_matrix = cv2.getRotationMatrix2D(image_center,10,1.0)
        rotated_image                                                  =
cv2.warpAffine(data_pixel_df_array,rotation_matrix,data_pixel_df_array.sh
ape)

        #ab=flipped_image
        ab = flipped_image.reshape(2304,)
        flipped_image_in_str = " ".join(str(a) for a in ab)
        emotion_list = np.append(emotion_list,flipped_image)
        image_list = np.append(image_list,rotated_image)

        #cd=rotated_image
        cd = rotated_image.reshape(2304,)
        rotated_image_in_str = " ".join(str(c) for c in cd)
        emotion_list = np.append(emotion_list,rotated_image)
        image_list = np.append(image_list,rotated_image)
        data_pixel_df=np.array([])
    augmented_dataframe = pd.DataFrame({
                    'emotion': emotion_list,
                    'pixels' : image_list
                    })
    return augmented_dataframe

augmented_dataset = data_augment("trainDataset.csv")
```

```
dataSet =  augmented_dataset

#separating augmented_dataset as per emotions
emotion_0 = dataSet[dataSet['emotion']==0]
emotion_1 = dataSet[dataSet['emotion']==1]
emotion_2 = dataSet[dataSet['emotion']==2]
emotion_3 = dataSet[dataSet['emotion']==3]
emotion_4 = dataSet[dataSet['emotion']==4]
emotion_5 = dataSet[dataSet['emotion']==5]
emotion_6 = dataSet[dataSet['emotion']==6]

del dataset #to free memory

#to balance out less amount of data in emotion_1 : Disgust
emotion_1 = pd.concat([emotion_1]*4,ignore_index=True)

#shuffling emotion_1
emotion_1=emotion_1.sample(frac=1).reset_index(drop=True)


#separating train and valid datasets as per emotions
emotion_0_valid , emotion_0_train = takeRandom(emotion_0, fraction=0.2)
emotion_1_valid , emotion_1_train = takeRandom(emotion_1, fraction=0.2)
emotion_2_valid , emotion_2_train = takeRandom(emotion_2, fraction=0.2)
emotion_3_valid , emotion_3_train = takeRandom(emotion_3, fraction=0.2)
emotion_4_valid , emotion_4_train = takeRandom(emotion_4, fraction=0.2)
emotion_5_valid , emotion_5_train = takeRandom(emotion_5, fraction=0.2)
emotion_6_valid , emotion_6_train = takeRandom(emotion_6, fraction=0.2)

#concating all the emotions into train and valid datasets
validDataset                                                            =
pd.concat([emotion_0_valid,emotion_1_valid,emotion_2_valid,emotion_3_v
alid,emotion_4_valid,emotion_5_valid,emotion_6_valid])
trainDataset                                                            =
pd.concat([emotion_0_train,emotion_1_train,emotion_2_train,emotion_3_tr
ain,emotion_4_train,emotion_5_train,emotion_6_train])
```

```
#shuffling train and valid datasets
trainDataset = trainDataset.sample(frac=1).reset_index(drop=True)
validDataset = validDataset.sample(frac=1).reset_index(drop=True)


validDataset.to_csv("validDataset.csv",index=False)
trainDataset.to_csv("trainDataSet.csv",index=False)



import csv
import numpy as np


pixel_depth = 255.0

def load_csv(filename,length,force=False):
    num = 0
    data = np.ndarray(shape=(length,28,28),dtype=np.float32)
    labels = np.ndarray(shape=(length,),dtype=np.float32)
    with open(filename) as csvfile:
        fer2013 = csv.reader(csvfile)
        for row in fer2013:
            if force == False:
                force = True
            else:
                d = row[1].split(" ")
                l = row[0]
                d = np.array(d,dtype=float).reshape((28,28))
                l = np.array(l,dtype=float)
                data[num,:,:] = d
                labels[num,]= l
                num = num + 1

    #normalized image dataset
    data = data[0:num,:,:]/255.0
    labels = labels[0:num,]
    print('Full dataset tensor:', data.shape)
    print('Label of dataset: ',labels.shape)
    print('Mean:', np.mean(data))
    print('Standard deviation:', np.std(data))
    return data,labels




import pandas as pd
train = pd.read_csv("trainDataset.csv")
valid = pd.read_csv("trainDataset.csv")
test = pd.read_csv("trainDataset.csv")
length_train = train.shape[0]
length_valid = valid.shape[0]
length_test = test.shape[0]




#loading train, valid and test dataset
train_dataset,train_labels = load_csv("trainDataset.csv",length_train)



test_dataset,test_labels = load_csv("testDataset.csv",length_test)



valid_dataset,valid_labels = load_csv("validDataset.csv",length_valid)




image_size = 28
num_labels = 7
num_channels = 1 # grayscale


#reformatting input data to make then tensorflow ready
def reformat(dataset, labels):
    dataset = dataset.reshape(
    (-1, image_size, image_size, num_channels)).astype(np.float32)
    labels = (np.arange(num_labels) == labels[:,None]).astype(np.float32)
    return dataset, labels
train_dataset, train_labels = reformat(train_dataset, train_labels)
valid_dataset, valid_labels = reformat(valid_dataset, valid_labels)
test_dataset, test_labels = reformat(test_dataset, test_labels)
print('Training set', train_dataset.shape, train_labels.shape)
print('Validation set', valid_dataset.shape, valid_labels.shape)
print('Test set', test_dataset.shape, test_labels.shape)



#calculating accuracy of train, validation and test datasets
def accuracy(predictions, labels):
    return (100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(labels,
1))
          / predictions.shape[0])



import tensorflow as tf



#hyperparameters
batch_size = 32
patch_size = 5
depth = 16
num_hidden = 64

#defining tensorflow graph
graph = tf.Graph()

with graph.as_default():

  # Input data : train dataset, validation dataset and test dataset
    tf_train_dataset = tf.placeholder(
      tf.float32, shape=(batch_size, image_size, image_size,
num_channels))
```

```
  tf_train_labels = tf.placeholder(tf.float32, shape=(batch_size,
num_labels))
  tf_valid_dataset = tf.constant(valid_dataset)
  tf_test_dataset = tf.constant(test_dataset)

 # Variables : Weights and biases of diffent layers in CNN
  layer1_weights = tf.Variable(tf.truncated_normal(
    [patch_size, patch_size, num_channels, depth], stddev=0.1))
  layer1_biases = tf.Variable(tf.zeros([depth]))
  layer2_weights = tf.Variable(tf.truncated_normal(
    [patch_size, patch_size, depth, depth], stddev=0.1))
  layer2_biases = tf.Variable(tf.constant(1.0, shape=[depth]))
  layer3_weights = tf.Variable(tf.truncated_normal(
    [image_size // 4 * image_size // 4 * depth, num_hidden], stddev=0.1))
  layer3_biases = tf.Variable(tf.constant(1.0, shape=[num_hidden]))
  layer4_weights = tf.Variable(tf.truncated_normal(
    [num_hidden, num_labels], stddev=0.1))
  layer4_biases = tf.Variable(tf.constant(1.0, shape=[num_labels]))

 # Model.
  def model(data):
      #layer1
      conv = tf.nn.conv2d(data, layer1_weights, [1, 1, 1, 1],
padding='SAME')
      hidden = tf.nn.relu(conv + layer1_biases)
      pool = tf.nn.max_pool(hidden, ksize = [1, 2, 2, 1], strides = [1, 2, 2, 1],
padding = 'SAME')

      #layer2
      conv = tf.nn.conv2d(pool, layer2_weights, [1, 1, 1, 1],
padding='SAME')
      hidden = tf.nn.relu(conv + layer2_biases)
      pool = tf.nn.max_pool(hidden, ksize = [1, 2, 2, 1], strides = [1, 2, 2, 1],
padding = 'SAME')

      shape = pool.get_shape().as_list()
      reshape = tf.reshape(pool, [shape[0], shape[1] * shape[2] * shape[3]])

      #fully-connected layer
      fc = tf.nn.relu(tf.matmul(reshape, layer3_weights) + layer3_biases)
```

```
        #output layer
        return tf.matmul(fc, layer4_weights) + layer4_biases

  # Training computation.
    logits = model(tf_train_dataset)
    loss = tf.reduce_mean(
         tf.nn.softmax_cross_entropy_with_logits(labels=tf_train_labels,
logits=logits))

  # Optimizer.
    optimizer = tf.train.GradientDescentOptimizer(0.05).minimize(loss)

  # Predictions for the training, validation, and test data.
    train_prediction = tf.nn.softmax(logits)
    valid_prediction = tf.nn.softmax(model(tf_valid_dataset))
    test_prediction = tf.nn.softmax(model(tf_test_dataset))



#no. of iterations
num_steps = 16001

#Session definition
with tf.Session(graph=graph) as session:
    tf.global_variables_initializer().run()
    print('Initialized')
    s = []
    mini_acc = []
    val_acc = []
    losses = []
    for step in range(num_steps):
        offset = (step * batch_size) % (train_labels.shape[0] - batch_size)
        batch_data = train_dataset[offset:(offset + batch_size), :, :, :]
        batch_labels = train_labels[offset:(offset + batch_size), :]
        feed_dict = {tf_train_dataset : batch_data, tf_train_labels :
batch_labels}
        _, l, predictions = session.run(
         [optimizer, loss, train_prediction], feed_dict=feed_dict)
        if (step % 2000 == 0):
            min_a = accuracy(predictions, batch_labels)
            val_a = accuracy(valid_prediction.eval(), valid_labels)
            s.append(step)
            mini_acc.append(min_a)
            val_acc.append(val_a)
            losses.append(l)
            print('Minibatch loss at step %d: %f' % (step, l))
            print('Minibatch accuracy: %.1f%%' % min_a)
            print('Validation accuracy: %.1f%%' % val_a)
    x = test_prediction.eval()
    print('Test accuracy: %.1f%%' % accuracy(x, test_labels))



#maximum probability value
a = np.argmax(x,1)

#actual test labels
b = np.argmax(test_labels,1)


#creating dataframe of predicted and actual labels
a_pd = pd.DataFrame({"a":a})
a_pd['b'] = b



#summing predicted labels
_0=np.sum(a_pd['a']==0)
_1=np.sum(a_pd['a']==1)
_2=np.sum(a_pd['a']==2)
_3=np.sum(a_pd['a']==3)
_4=np.sum(a_pd['a']==4)
_5=np.sum(a_pd['a']==5)
_6=np.sum(a_pd['a']==6)
```

```
#summing actual labels
b_0=np.sum(a_pd['b']==0)
b_1=np.sum(a_pd['b']==1)
b_2=np.sum(a_pd['b']==2)
b_3=np.sum(a_pd['b']==3)
b_4=np.sum(a_pd['b']==4)
b_5=np.sum(a_pd['b']==5)
b_6=np.sum(a_pd['b']==6)



#list in the form [True,False,False,.....] for predicted values
ind_0a =(a_pd['a']==0)
ind_1a =(a_pd['a']==1)
ind_2a =(a_pd['a']==2)
ind_3a =(a_pd['a']==3)
ind_4a =(a_pd['a']==4)
ind_5a =(a_pd['a']==5)
ind_6a =(a_pd['a']==6)



#list in the form [True,False,False,.....] for actual values
ind_0b =(a_pd['b']==0)
ind_1b =(a_pd['b']==1)
ind_2b =(a_pd['b']==2)
ind_3b =(a_pd['b']==3)
ind_4b =(a_pd['b']==4)
ind_5b =(a_pd['b']==5)
ind_6b =(a_pd['b']==6)


#for calculating true positives
def accu(a,b):
    sum = 0
    for i in range(len(a)):
        if b[i]:
            if a[i]==b[i]:
                sum+=1
    return sum
```

```
acc_0 = accu(ind_0a,ind_0b)
acc_1 = accu(ind_1a,ind_1b)
acc_2 = accu(ind_2a,ind_2b)
acc_3 = accu(ind_3a,ind_3b)
acc_4 = accu(ind_4a,ind_4b)
acc_5 = accu(ind_5a,ind_5b)
acc_6 = accu(ind_6a,ind_6b)



print("categories                    (0=An,  1=Dis,  2=Fe,  3=Ha,  4=Sa,  5=Sup,
6=Neutral).")
print(" actual labels count  ",b_0,b_1,b_2,b_3,b_4,b_5,b_6)
print("predicted labels count",_0,_1,_2,_3,_4,_5,_6)
print("true positives        ",acc_0,acc_1,acc_2,acc_3,acc_4,acc_5,acc_6)

#no. of classes
n_groups = 7

index = np.arange(n_groups)
bar_width = 0.3
opacity = 0.8

#for plotting accuracy graphs
rects1 = plt.bar(index,actual,bar_width,alpha = opacity, color = 'b',
label='Actual')
rects1 = plt.bar(index+bar_width,predicted,bar_width,alpha = opacity, color
= 'g', label='Predicted')
rects1 = plt.bar(index+2*bar_width,correctPredicted,bar_width,alpha =
opacity, color = 'y', label='Accuracy')
plt.xlabel('Emotions')
plt.ylabel('Values')
plt.title('Predicted vs Actual')
plt.xticks((index+bar_width),(0,1,2,3,4,5,6))
plt.legend()

plt.tight_layout()
plt.show()

plt.plot(s, mini_acc)
plt.plot(s, valid_acc)
plt.legend(['mini_acc','val_acc'])
plt.title('Minibatch and Validation accuracy VS Steps')
plt.show()

plt.plot(s,losses)
plt.legend(['losses'])
plt.title('Loss VS Steps')
plt.show()
```

OUTPUT

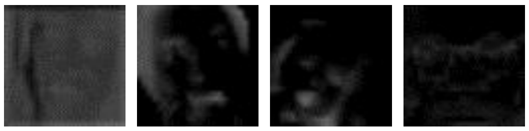
```
dbda@ACTS66:~/dataset/Code>python
kaggle_img_tensor_ready_16000iter_batch16_withPool.py

Full dataset tensor: (57657, 28, 28)
Label of dataset:  (57657,)
Mean: 0.49015
Standard deviation: 0.249404
Full dataset tensor: (6576, 28, 28)
Label of dataset:  (6576,)
Mean: 0.500332
Standard deviation: 0.245935
Full dataset tensor: (6177, 28, 28)
Label of dataset:  (6177,)
Mean: 0.489069
Standard deviation: 0.249652
Training set (57657, 28, 28, 1) (57657, 7)
Validation set (6177, 28, 28, 1) (6177, 7)
Test set (6576, 28, 28, 1) (6576, 7)
Initialized
Minibatch loss at step 0: 3.693719
Minibatch accuracy: 9.4%
Validation accuracy: 25.3%
Minibatch loss at step 2000: 1.719000
Minibatch accuracy: 28.1%
Validation accuracy: 36.2%
Minibatch loss at step 4000: 1.451431
Minibatch accuracy: 50.0%
Validation accuracy: 36.9%
Minibatch loss at step 6000: 1.347095
Minibatch accuracy: 43.8%
Validation accuracy: 41.0%
Minibatch loss at step 8000: 1.396661
Minibatch accuracy: 50.0%
Validation accuracy: 42.9%
Minibatch loss at step 10000: 1.462906
Minibatch accuracy: 50.0%
Validation accuracy: 43.9%
Minibatch loss at step 12000: 1.244038
Minibatch accuracy: 56.2%
Validation accuracy: 45.1%
Minibatch loss at step 14000: 1.475517
Minibatch accuracy: 43.8%
Validation accuracy: 47.2%
Minibatch loss at step 16000: 1.234785
Minibatch accuracy: 56.2%
Validation accuracy: 47.3%
Test accuracy: 48.8%


sum of Predicted values with respect to emotion
444 187 636 1956 1534 935 884

sum of Actual values with respect to emotion
910 94 910 1721 1136 653 1152

acuracy with respect to emotions
categories          (0=An, 1=Dis, 2=Fe, 3=Ha, 4=Sa, 5=Sup, 6=Neutral).
 actual labels count       910 94 910 1721 1136 653 1152
predicted labels count 444 187 636 1956 1534 935 884
True positives              200 63 220 1297 543 448 449
```

VISUALIZATION

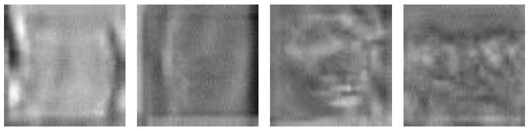
Feature Map of Convolution layer – 1

Feature Map of Hidden layer - 1

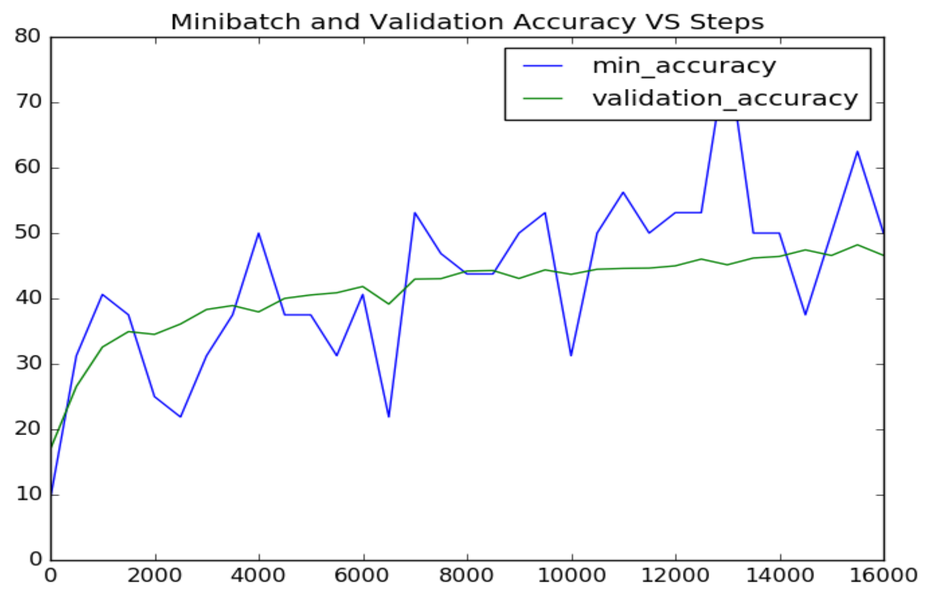
Feature Map of Convolution layer – 2



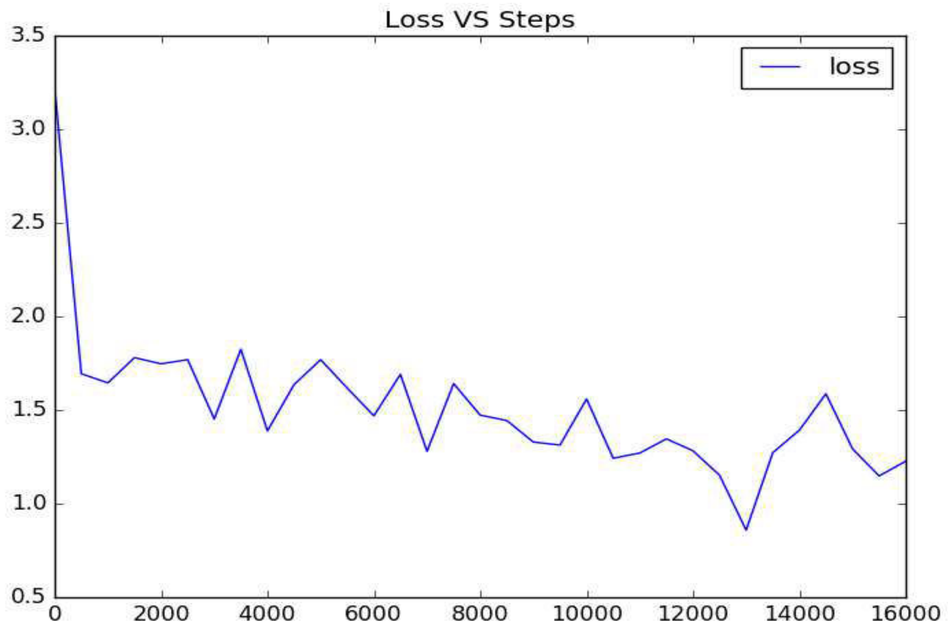
Feature Map of Hidden layer – 2



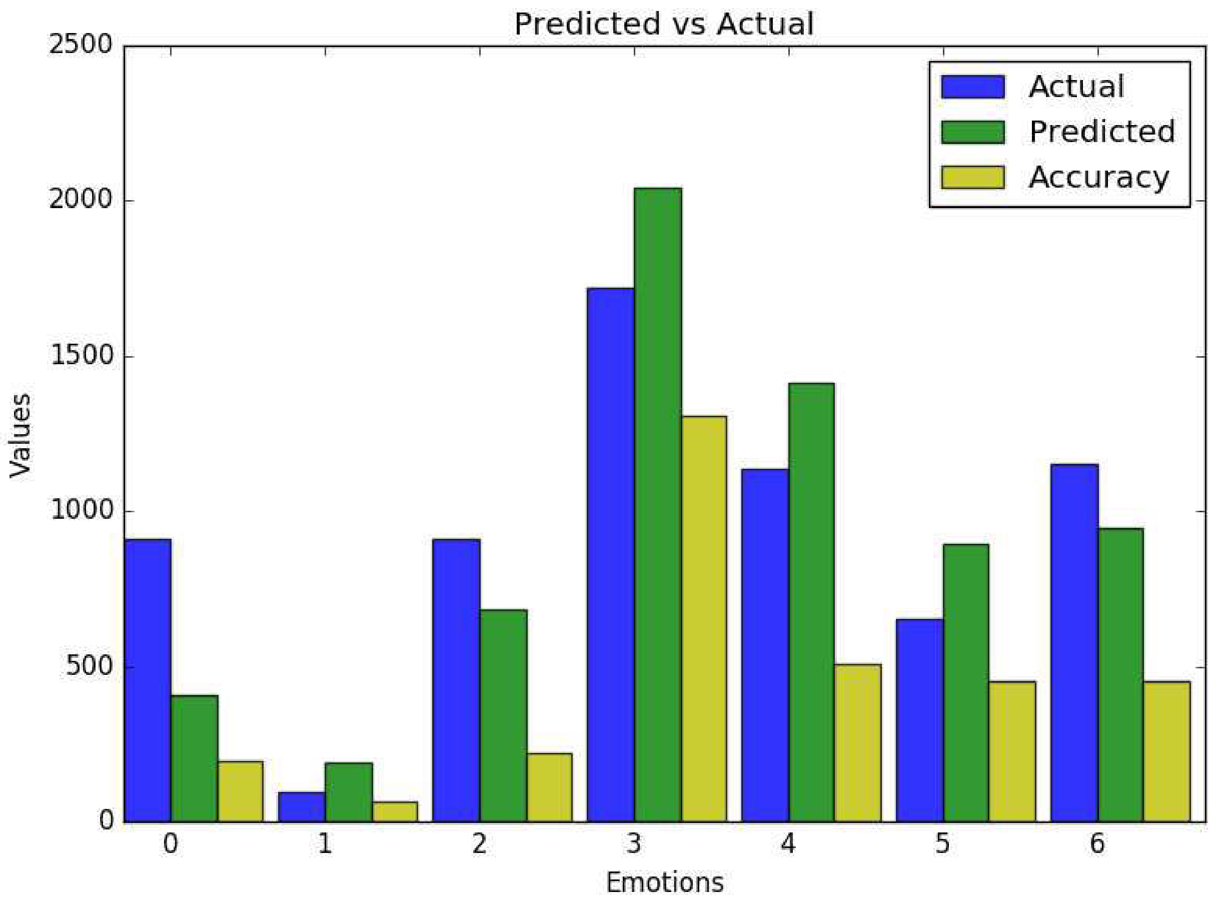
Plot of Minibatch Accuracy , Validation Accuracy VS Steps



Plot of Loss VS Steps



Bar chart of Actual, Predicted and True positives



CONCLUSION

Each image was classified into seven classes and the probability of the image lying in each class was calculated and shown in the output. We used pySpark for batch processing.

We observed that the loss function has a decreasing trendline and we get the least value at around 13000 epoch. Anger performed least accurately. The overall accuracy came out to be 49.0 %.

BIBLIOGRAPHY

- Hsiao Chen Chang, Emilien Dupont, William Zhang (March 13, 2016) CS231N Project: Facial Expression Recognition, Stanford University, CS231N Final Report.
- Adesh Pandey : https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/
- Andre Teixeira Lopes, Edilson de Aguiar, Alberto F. De Souza, Thiago Oliveira-Santos; Facial Expression Recognition with Convolutional Neural Networks: Coping with Few Data and the Training Sample Order
- Wikipedia : https://en.wikipedia.org/wiki/Convolutional_neural_network
- Wikipedia : https://en.wikipedia.org/wiki/Artificial_neural_network
- https://www.tensorflow.org/