Capstone Report for Machine Learning Advanced Nanodegree

Venkatesh Prasad V M

ASL Alphabets Recognition

11th July 2018

I. DEFINITION

PROJECT OVERVIEW

People suffering from speaking disability use sign language as a way of communication like ASL (American Sign Language). The problem is, it is not understood by common population and this isolates them from the society. Now a days computers are really good at understanding human communications, even better than other humans. There are some existing methods which detects with Computer vision (referred here), HSV color model and edge detection (referred here) and surface EMG (referred here). From all these methods deep learning is found to be more accurate and cost effective.

In this project, I created an application with python and keras (an ML library for python) to build a CNN (Convolutional Neural Network) model which predicts the alphabet from the given ASL alphabet image as input and prints the result.

PROBLEM STATEMENT

To determine whether a hand gesture image is an American Sign Language alphabet and find the appropriate class label (or alphabet) from the image.

MERTICS

The Convolutional Neural Network is used to solve this problem. The CNN will learn the important features of the image, in this case number of opened fingers, angle of a finger, finger grooves, etc., and classify the image to the corresponding class.

The loss function used is categorical_crossentropy since the classification for my model is multiclass classification and ‘binary_crossentropy’ is not suitable. The loss function is given by,

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C}\mathbf{1}_{y_i \in C_c}\log p_{model}[y_i \in C_c]$$

whose number is N, and the categories c, whose number is C. The term 1_{y_i \in C_c} is the indicator function of the i^{th} observation belonging to the c^{th} category. The p_{model}[y_i \in C_c] is the probability predicted by the model for the i^{th} observation to belong to the c^{th} category (Referred from here).

Adam optimizer is used here to update weights and bias. This is more suitable as Adam combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems like blurry images. It is also easy to configure. More information about Adam optimizer can be found here.

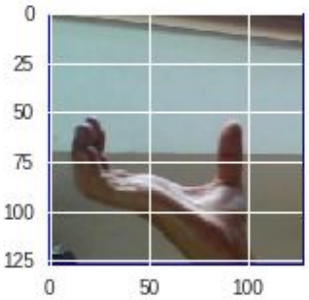
II. ANALYSIS

DATA EXPLORATION

The dataset is taken from Kaggle. The dataset contains two zips, asdfsadf.zip as training data and afdasdf.zip as test data. The training data set contains 87,000 images, from which 0.1 fraction is used for validation data. The shape of every image is (200, 200, 3). There are 29 classes, of which 26 are for the letters A-Z and 3 classes for SPACE, DELETE and NOTHING. These 3 classes are very helpful in real time applications, and classification. The test data set contains 29 images to encourage the use of real world test images.

```
Labels : ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'del',
'nothing', 'space']
```

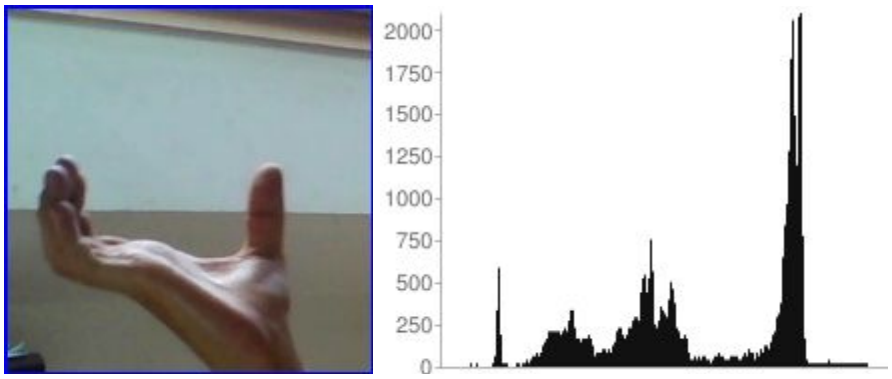
Below is a sample image from ‘SPACE’ label folder which is plotted in 128x128 resolution.



```
Label: 'space'
Shape: (128, 128, 3)
```

EXPLORATORY VISUALIZATION

The distribution of data is a is equal for every class (or label) since all the folders has equal number of images which is 3000. The bar graph of this distribution will be same for all the labels and hence it is not shown here. Instead we can see the grayscale histogram of an image from set.



The left image is the 'SPACE' labeled image and the right image is the grayscale histogram of the right image. Histogram is based on the frequency of luminance in the image. The luminance is computed for each pixel with the formula 0.3*R+0.59*G+0.11*B. This intensity distribution of the image helps the model the identify the color of the hand and differentiate it from the background color to make accurate classification.

ALGORITHMS AND TECHNIQUES

This problem is solved using the Convolutional Neural Network or Covnet architecture. A brief explanation of this covnet architecture is given below from a course notes by Stanford University.

"Convolutional Neural Networks are very similar to ordinary Neural Networks and they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for learning regular Neural Networks still apply.

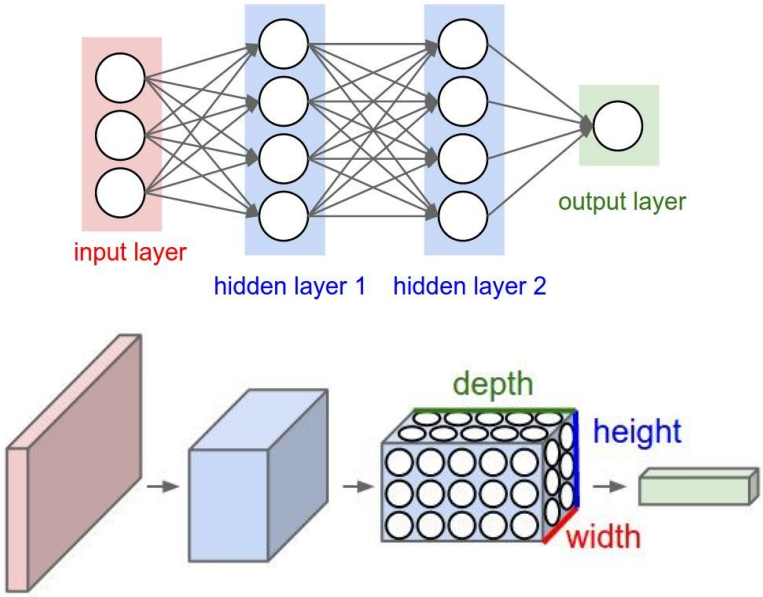
So what does change? ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

Architecture Overview

Neural Networks receive an input (a single vector), and transform it through a series of hidden layers. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer, and where neurons in a single layer function completely independently and do not share any connections. The last fully-connected layer is called the "output layer" and in classification settings it represents the class scores.

*Regular Neural Nets don't scale well to full images. In CIFAR-10, images are only of size 32x32x3 (32 wide, 32 high, 3 color channels), so a single fully-connected neuron in a first hidden layer of a regular Neural Network would have 32*32*3 = 3072 weights. This amount still seems manageable, but clearly this fully-connected structure does not scale to larger images. For example, an image of more respectable size, e.g. 200x200x3, would lead to neurons that have 200*200*3 = 120,000 weights. Moreover, we would almost certainly want to have several such neurons, so the parameters would add up quickly! Clearly, this full connectivity is wasteful and the huge number of parameters would quickly lead to overfitting.*

3D volumes of neurons. Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: ***width, height, depth****. (Note that the word depthhere refers to the third dimension of an activation volume, not to the depth of a full Neural Network, which can refer to the total number of layers in a network.) For example, the input images in CIFAR-10 are an input volume of activations, and the volume has dimensions 32x32x3 (width, height, depth respectively). As we will soon see, the neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. Moreover, the final output layer would for CIFAR-10 have dimensions 1x1x10, because by the end of the ConvNet architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension. Here is a visualization:*



First: A regular 3-layer Neural Network. Second: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

A ConvNet is made up of Layers. Every Layer has a simple API: It transforms an input 3D volume to an output 3D volume with some differentiable function that may or may not have parameters.

Layers used to build ConvNets

As we described above, a simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures: ***Convolutional Layer****,* ***Pooling Layer****, and* ***Fully-Connected Layer****. We will stack these layers to form a full ConvNet* ***architecture****."*

BENCHMARK

For benchmark, I found a model submitted by a kaggler for the competition, that predicts the appropriate alphabet with an accuracy of 91% which runs on 10 epochs, uses both adam and rmsprop optimizer with learning rate 0.0001. The objective of taking this model is to show that

the adam optimizer alone is enough and it can be trained efficiently with less number of parameters.

III. METHODOLOGY

DATA PREPROCESSING

Preprocessing of data is the most crucial part in CNN before building a model as this determines the accuracy of the model and the steps taken for data preprocessing in this project are as follows,

- ➔ The training set and test set is downloaded from Kaggle using Kaggle CLI and extracted. The test set is never touched while creating the model to make it unbiased while testing.
- ➔ All the 200x200 images are resized to 128x128 images which reduces the unnecessary detailing in the image background and also reduces the time taken to train from the dataset.
- ➔ The fraction 0.15 from the dataset is taken as validation set and the remaining is taken as the training set since this fraction helps the validation process while training the model.
- ➔ For every image, rotation range of 20° and shift range of 0.2 is set. This will help the model to recognize the hand even if it is slightly off from the actual position and angle which eventually increases the accuracy of the model.

IMPLEMENTATION

Google Colab is used for implementing this is project. It offers a customized jupyter notebook which is powered by their own GPUs. You can use this medium blog to learn how to use Google Colab. Many deep learning libraries like Tensorflow, Keras are pre-installed in this. The dataset was downloaded into the runtime and extracted.

This data is then preprocessed and then, split into training and validation set using ImageDataGenerator class in Keras. The complications like overfitting, underfitting, time taken to train the model were solved during implementation by using trial and error method of changing the hyperparameters. The CNN architecture of this project is,

- ➔ Layer 1 : inputs = 64, kernel_size=(3, 3), strides=1
- ➔ inputs = 64, kernel_size=(3, 3), strides=2
- ➔ Dropout: probability = 0.5
- ➔ Layer 2 : inputs = 128, kernel_size=(3, 3), strides=1

- ➔ inputs = 128, kernel_size=(3, 3), strides=2
- ➔ Dropout: probability = 0.5
- ➔ Layer 3 : inputs = 256, kernel_size=(3, 3), strides=1
- ➔ inputs = 256, kernel_size=(3, 3), strides=2
- ➔ Dropout: probability = 0.5
- ➔ Flattening the layer.
- ➔ Dense : inputs = 512
- ➔ Output layer is fully connected layer with softmax activation function.
- ➔ The model is compiled with Adam optimizer and 'categorical_crossentropy' as the loss function.
- ➔ The model is trained for 6 epochs with batch size of 128.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 126, 126, 64)      1792
_________________________________________________________________
activation_1 (Activation)    (None, 126, 126, 64)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 62, 62, 64)        36928
_________________________________________________________________
activation_2 (Activation)    (None, 62, 62, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 62, 62, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 60, 60, 128)       73856
_________________________________________________________________
activation_3 (Activation)    (None, 60, 60, 128)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 29, 29, 128)       147584
_________________________________________________________________
activation_4 (Activation)    (None, 29, 29, 128)       0
_________________________________________________________________
dropout_2 (Dropout)          (None, 29, 29, 128)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 27, 27, 256)       295168
_________________________________________________________________
activation_5 (Activation)    (None, 27, 27, 256)       0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 13, 13, 256)       590080
_________________________________________________________________
activation_6 (Activation)    (None, 13, 13, 256)       0
_________________________________________________________________
```

```
flatten_1 (Flatten)          (None, 43264)             0
_________________________________________________________________
dropout_3 (Dropout)          (None, 43264)             0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               22151680
_________________________________________________________________
dense_2 (Dense)              (None, 29)                14877
=================================================================
Total params: 23,311,965
Trainable params: 23,311,965
Non-trainable params: 0
_________________________________________________________________
```

REFINEMENT

Initial result is obtained by building a simple CNN architecture and evaluated. This resulted in more loss (around 0.6) and less accuracy (around 0.8). After the proper tuning of the hyperparameters for a number of times the model's accuracy increased. For instance, layers inputs were changed from (16, 32, 48), (128, 64, 32) and finally to (64, 128, 256), dropout layer was added after every layer for avoiding overfitting, the number of epochs was reduced from 10 to 6 because the accuracy remained the same after the 6th epoch. After performing all the refinements the accuracy increased to 93%, the exact value was `0.9332` and the loss was drastically reduced to `0.1965`.

IV. RESULTS

MODEL EVALUATION AND VALIDATION

As discussed in the data preprocessing section a fraction `0.15` is used as validation set from the original dataset. The test set was downloaded seperately which was provided in the kaggle. After training with the validation set the final accuracy was `0.9332`.

Since the test set untouched during the training of the model, it is totally new and unseen by the model and thus the classification is unbiased. Moreover the test set also contained images of hands signs in different backgrounds and were classified well. This implied that the model can be trusted and it is not sensitive even if there is significant changes in the data. The status of the training at the last epoch was,

```
578/578 [==============================] - 567s 980ms/step - loss: 0.1965 -
acc: 0.9332 - val_loss: 0.4978 - val_acc: 0.8457
```

This implies that the final accuracy is 0.9332 and the validation accuracy is 0.8457.

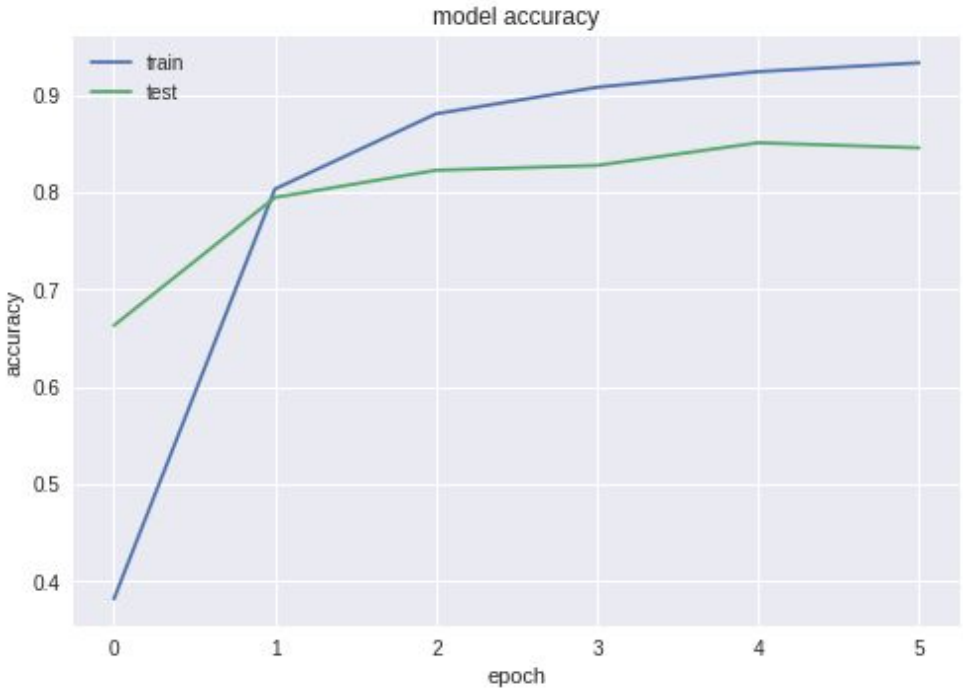
JUSTIFICATION

After thorough analyzation of the result obtained from the model, it is realised that this accuracy crossed the accuracy of the model discussed in benchmark by 2%. The accuracy of the benchmark model is around 90% and the accuracy of my model is around 93%. This accuracy is significant to solve the problem which is again verified by the test set which was separately provided in the kaggle.

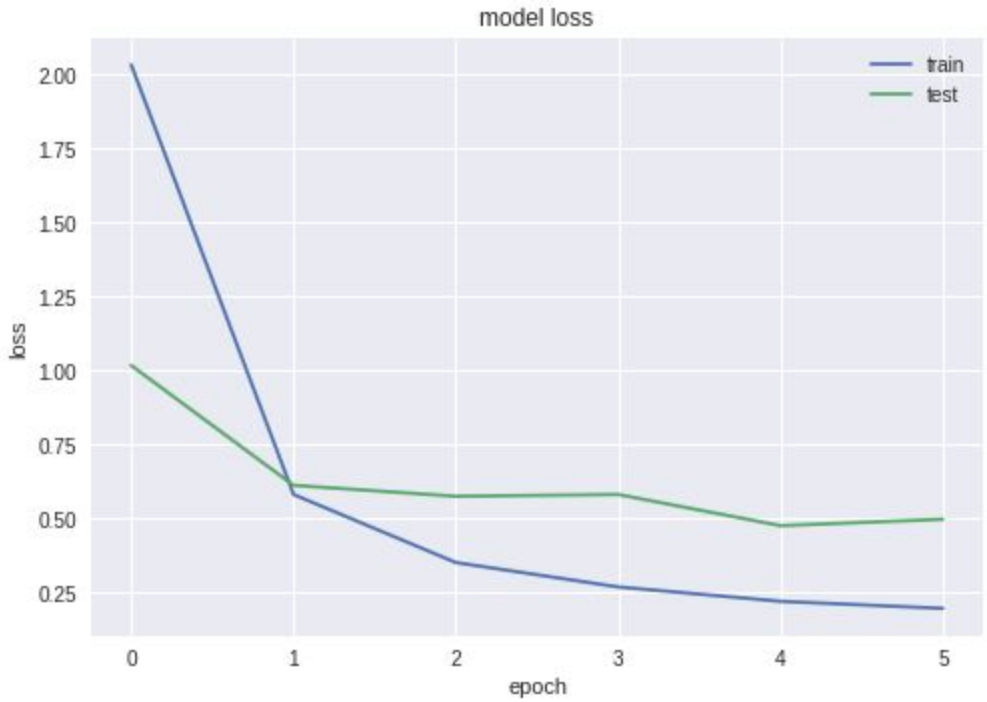
V. CONCLUSION

FREE-FORM VISUALIZATION

The accuracy per epoch is plotted in the model accuracy graph which is shown below. It is clearly seen that the accuracy in increases in every epoch. The graph shows both train accuracy (in blue color) and test accuracy (in orange color).



The loss per epoch is plotted as model loss graph which is shown below. The loss is reduced in every epoch and the loss function used is ‘categorical_crossentropy’ loss function. The graph shows both train loss (in blue color) and test loss (in orange color).



REFLECTION

The process used in the process can be summarised as,

1. Google Colab was used to implement the project.
2. The dataset was downloaded in to the colab runtime by kaggle CLI.
3. This dataset was split into training and verification set and pre-processed.
4. Hyperparamteres were set to appropriate values to get higher accuracy.
5. Using keras library a CNN model was built.
6. Adam optimizer as optimizer and categorical_crossentropy as loss function is used to train the model.
7. The model achieved an accuracy of 93.32% of accuracy.

I found the step 4 more important and more crucial since it took many iterations of refinement to get better accuracy. During the preprocessing of image, I first tried with 64x64 images and I got good accuracy but the loss was really high which made me learn that pixel informations are more important. Later on I increased the resolution to 128x128 and the loss was reduced significantly.

Google colab GPU really helped in reducing the time taken for training and made all the iterations of refinement simple.

IMPROVEMENT

To make this process user friendly this project can be converted to an application say android app. There are two methods to make an android application with this model.

1. By saving the keras model and convert it to tensorflow lite model and deploy it directly in the android app. (But this will drastically increase the apk size).
2. By converting the keras model to tensorflow lite model and hosting the model in firebase with the help of the new ML kit.