SAN JOSÉ STATE UNIVERSITY

Fall 2019
EE258: Neural Networks

Under the guidance of Prof. Birsen Sirkeci

KANNADA MNIST

Aniket Phatak (012564994)
Dhananjay Bajpai (013851812)

Abstract:

Kannada MNIST is composed of 60,000 images and includes 10 classes of numbers from 0-9 with a dimension of 28x28 pixels. Our aim is to build a neural network and train it with the Kannada MNIST Dataset. Our implementation mainly deals with the utilization of optimum parameters to tweak a Convolutional Neural Network such that highest performance is achieved. In our current implementation, we have CNN with several tweaks by changing thing such as number of convolutional layers, varying Batch size, varying the number of epoch's , changing the activation functions in the output layer, varying number of neurons in the flatten fully connected layer, varying momentum during Batch normalization , changing the optimizer settings , varying the loss losses and number of filters in the convolutional layer are some of our interests that we deal with so as to obtain the highest performing network after making necessary changes. Parameter tweaks are done through Keras API and corresponding results are obtained. A proper intuition of which network to be used on a certain type of data is a critical Deep Learning attribute that we must know. To achieve this, we must understand the behavior of data such that we can find out if a particular classification network is ideal for that application to obtain maximum possible accuracy for our implementation which is to classify 10 image classes. The evaluation metric for this project is the categorization accuracy or the proportion of test images that are correctly classified.

Approach:

1) **Import all necessary libraries**
- For our implementation we utilize Keras to build our CNN. We also make use of NumPy, Seaborn, Matplotlib & Pandas for data visualization & array manipulation.

2) **Import the dataset**
- We obtain the Kannada MNIST dataset from Kaggle Competition page and obtain the train and test files which are later stored in NumPy arrays

3) **Explore & Visualize the data to get an intuition of various attributes.**
- To get a proper intuition of our data we apply several built-in methods from Keras to get the dimensions and label size for both the training as well as test dataset.

4) **Data Preprocessing**
- This involves certain techniques such as data normalization such that our images fall within the range of 0 to 1. The reason for normalization is that larger values should not dominate the feature of a classifier

5) **Prediction for CNN with several parameter tweaks.**
- Simulations for various changed parameters such as number of Epoch's, Activation Functions, Learning rate, Optimizers, Batch size, Loss functions etc. In addition to this we also perform several changes in the convolutional layer itself such as tweaking dropout values after each convolution, adding/removing convolutional layer etc. are played with.

6) **Comparison of performance metrics of different networks**
- Results for model loss, accuracy for training, test and validation set where our model is subjected to various design tweaks.

Dataset: (28x8 labelled Kannada hand drawn digits from 0 through 9)

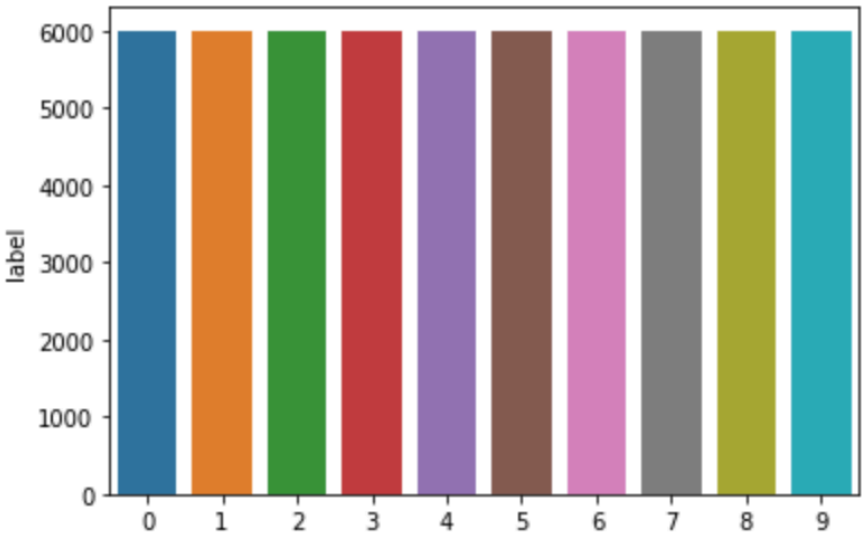
In this project we use Kannada MNIST data provided on Kaggle. The data files train and test consist of grayscale images of hand drawn digits from zero through nine in the Kannada script. Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255, inclusive.

The training dataset has 785 columns. The first column called digit is the one that is being drawn by the user. The rest of the values are pixel values of the associated image. Each pixel column in the training set has a name like pixel{x}, where x is an integer between 0 and 783, inclusive. To locate this pixel on the image, suppose that we have decomposed x as x = i * 28 + j, where i and j are integers between 0 and 27, inclusive. Then pixel{x} is located on row i and column j of a 28 x 28 matrix, (indexing by zero)

The pixel value makes up the image which is illustrated below:

```
000 001 002 003 ... 026 027
028 029 030 031 ... 054 055
056 057 058 059 ... 082 083
 |   |   |   |  ...  |   |
728 729 730 731 ... 754 755
756 757 758 759 ... 782 783
```

- All 9 classes of images are equally distributed among the train dataset file i.e. 6000 images of each class.

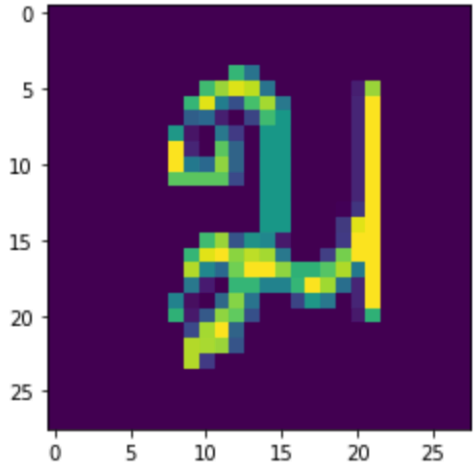


Pre-Processing

We perform several data enhancement techniques on our images in the form of preprocessing to make it robust with respect to predictions. The steps essentially involve image rotation, width and height shift, horizontal flip and changing zoom range.
We also perform one hot encoding to transform the categorical integer features into an appropriate form that can be fed to our model to do a better job in classification. Moreover, we also perform image transformation or normalization to get our pixel values in the range of [0,1].
Neural networks process inputs using small weight values, and inputs with large integer values can disrupt or slow down the learning process. As such it is good practice to normalize the pixel values so that each pixel value has a value between 0 and 1.

It is valid for images to have pixel values in the range 0-1 and images can be viewed normally. This can be achieved by dividing all pixel's values by the largest pixel value; that is 255. This is performed across all channels, regardless of the actual range of pixel values that are present in the image.

For example, after all these transformations image number 45 in our train images dataset looks like the below picture:



In order to avoid overfitting problem, we need to expand our handwritten digit dataset artificially. We can make existing dataset even larger. The idea is to alter the training data with small transformations to reproduce the variations occurring when someone is writing a digit.
For example, the number if not centered the scale is not the same (some who write with big/small numbers) The image is rotated is thus rotated. Approaches that alter the training data in ways that change the array representation while keeping the label same are known as data augmentation techniques. Some popular augmentations are grayscales, horizontal flips, vertical flips, random crops, color jitters, translations, rotations, and much more. By applying just, a couple of these transformations to our training data, we can easily double or triple the number of training examples and create a very robust model.

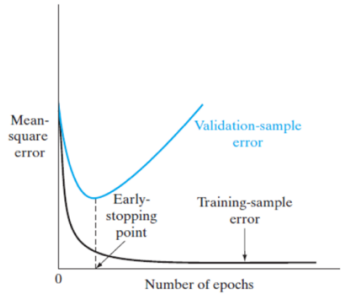
Some important Considerations while developing any neural network (MLP in our case):

- **Cross validation**

Cross validation technique is necessary to avoid the overfitting to the data while training the model. It assures the good generalization and improve performance of our model. To avoid overfitting condition in our Kannada MNIST dataset we split training data further into train & validation

- **Early Stopping**

When we start training the model both training, and the test errors start decreasing up to a certain point. If we continue to train our model it will start over fitting to the training data, so the training error continues to decrease but the generalization error starts increasing due to overfitting. So, to prevent this we should stop training the model after this point called the early stopping point. This can be illustrated from the figure below:



- **Confusion Matrix**

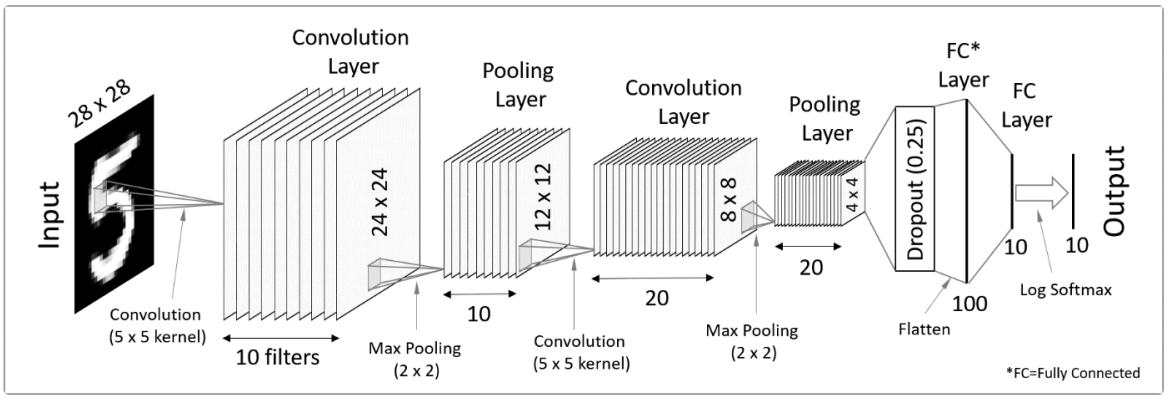
Confusion Matrix gives us insight not only into the misclassifications being made by a classifier but also the types of errors that are being made. Thus, Confusion matrix can be pretty useful when evaluating multiclass classifications. We can see which classes are are misclassified by our model clearly. Thus, a confusion matrix consists of summary of probable predictions and mainly used for error analysis to find False positives and True Positives in our classification problem. Correct classification data is always available at the diagonal while non diagonal elements show misclassified data i.e. one class represented falsely into some other classification label. We are using it for error analysis. We get correctly classification data at the diagonal.

- **One hot Encoding**

One hot encoding can be described as a methodology in which categorical variables are transformed into a form that can be fed to Machine Learning tasks to do a better job in prediction. Since the output of our model is going to show the probabilities of where an image should be categorized as a prediction. Each element represents the predicting probability of each classes. In our project, Keras inbuild library to_categorical () is used to do the on-hot encoding.

Convolutional Neural Network:

Since we are concerned with image classification, we utilize a convolutional neural network. The below diagram depicts CNN architecture.

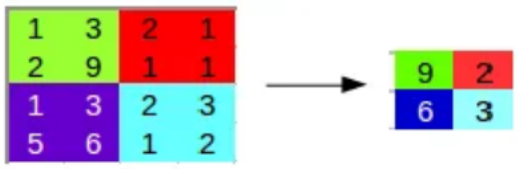


Padding: We can pad an image with an additional border i.e. by adding one pixel around the edges. This means that applying a convolution of 3x3 on it will result in a 6x6 matrix which is the original shape of the image. This is where padding comes into picture. This can be explained below:

```
Input: n X n
Padding: p
Filter size: f X f
Output: (n+2p-f+1) X (n+2p-f+1)
```

There are 2 main choices for padding. Valid: no padding. If we use valid padding, the output will be (n-f+1) X (n-f+1). Same: Here we apply padding so that the output size is the same as input size i.e. n+2p-f+1=n so p=f-1/2.

Pooling Layers: These are used to reduce the size of the inputs and hence speed up computation. Consider the 4x4 matrix shown below. Applying a max pooling operation on this matrix will result in a 2x2 output.



Thus, usually the CNN architecture involves a combination of conv layers and pooling layers at the beginning and a few fully connected layers at the end along with a SoftMax classifier for

classification. There are many hyperparameters that we tweak in this project so as to obtain the best performing model (discussed later)

Model: CNN

Default Network Architecture Settings:

1. Number of filters for the first 3 conv2D layers: 64
2. Number of filters for the next 3 conv2D layers: ----
3. Number of filters for the last conv2D layers: -----
4. number of neurons in the fully connected layer: 256-256-512-10
5. activation functions in the output layer: SoftMax
6. optimizer: Adelta
7. loss function: Categorical Cross Entropy
8. number of Convolutional Layer: 3
9. Dropout after each Convolution: --------
10. Batch Size: 1024
11. Momentum: 0.9
12. kernel size for the Convolution: 3-3-5
13. Pool size for Max pooling: 2x2
14. number of Epoch's: 2

Total Number of Epoch's = 2 for below result hence the lower accuracy

```
Epoch 1/2
58/58 [==============================] - 47s 817ms/step - loss: 13.8659 - accuracy: 0.6448 - val_loss: 4.0742 - val_accuracy: 0.1000
Epoch 2/2
58/58 [==============================] - 42s 718ms/step - loss: 0.8476 - accuracy: 0.9394 - val_loss: 0.7131 - val_accuracy: 0.9547
```

Training Accuracy	Validation Accuracy	Training Loss	Validation Loss
0.93	0.95	0.84	0.71

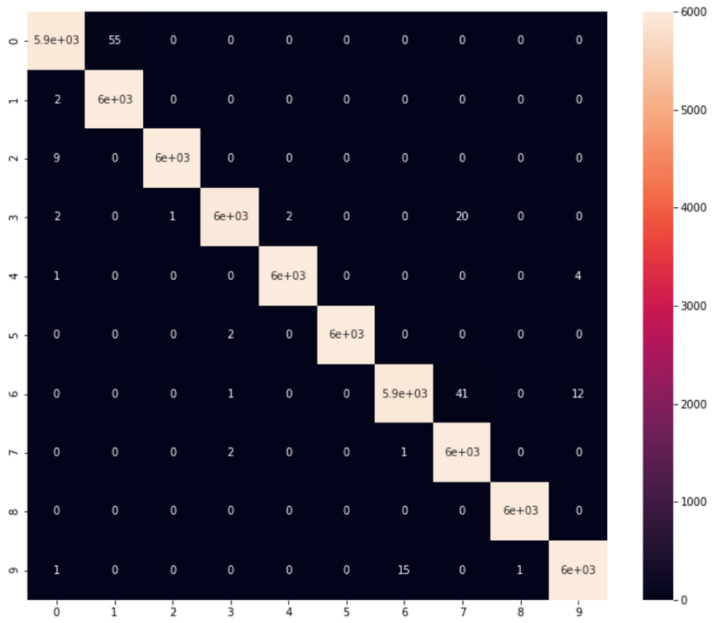
The difference between the training accuracy and test/validation accuracy can be called as overfitting. It happens when a ML model behaves worse on unseen data, i.e. the one it did not encounter in the training phase.

However, in this case since our epoch's are really low (just epoch=2) we will see that the training accuracy keeps improving and after a certain point it is better than the validation accuracy. At epoch's greater than 10 we can clearly notice that the training accuracy consistently keeps performing better and slightly worse on the unseen validation data

Confusion matrix for the above default setup can be represented as follows: Confusion matrix is mainly used for error analysis to find False positives and True Positives in our classification problem. We get correctly classification data at the diagonal.

	0	1	2	3	4	5	6	7	8	9
0	5945	55	0	0	0	0	0	0	0	0
1	2	5998	0	0	0	0	0	0	0	0
2	9	0	5991	0	0	0	0	0	0	0
3	2	0	1	5975	2	0	0	20	0	0
4	1	0	0	0	5995	0	0	0	0	4
5	0	0	0	2	0	5998	0	0	0	0
6	0	0	0	1	0	0	5946	41	0	12
7	0	0	0	2	0	0	1	5997	0	0
8	0	0	0	0	0	0	0	0	6000	0
9	1	0	0	0	0	0	15	0	1	5983

The heatmap representation for the same:

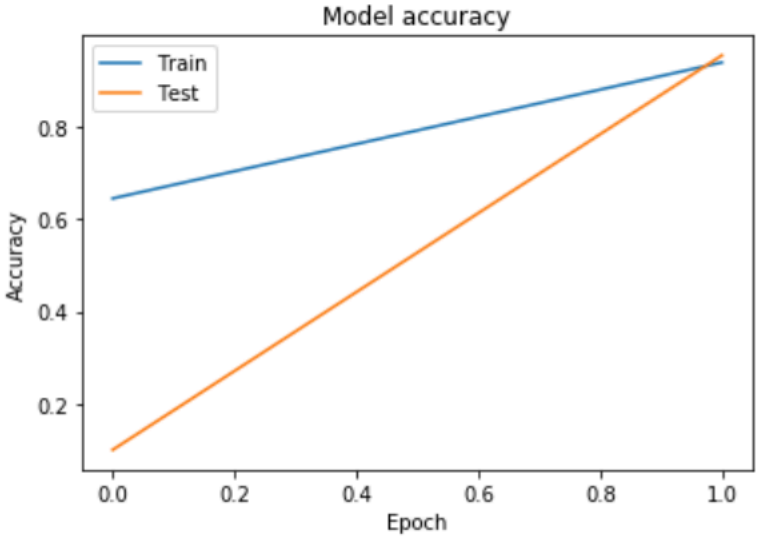


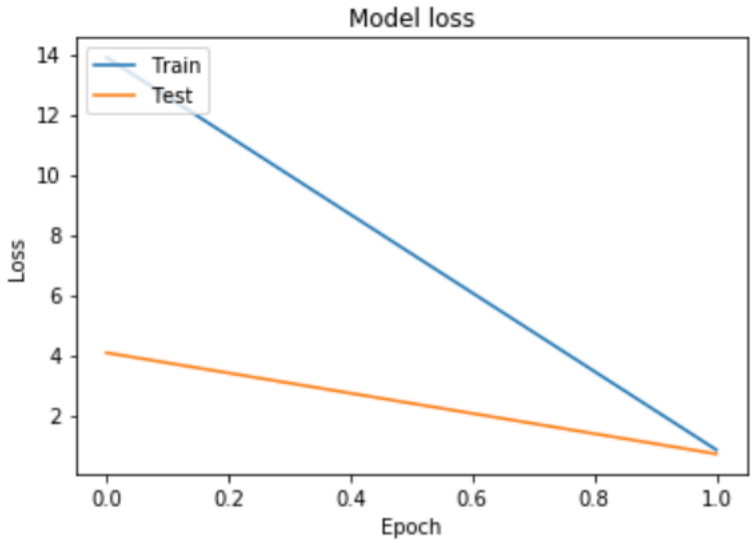
Parameters:

```
================================:
Total params: 2,783,882
Trainable params: 2,780,170
Non-trainable params: 3,712
_________________________________
```

Graphs for Loss and Accuracy:

Epochs=2





Changing the Batch Size:

Affects rate of learning

Batch size is one of the most important hyperparameters to tune in neural network systems. Experts often want to use a larger batch size to train their model as it allows computational fast processing mechanisms using the parallelism of GPUs. Although, it is also true that too big of a batch size will contribute to poor generalization.

Batch size directly dictates the accuracy of the rough calculation of the error gradient. This is done during the training phase of neural network. Minibatch gradient descent, Stochastic & Batch, and are the 3 main variations of this learning algorithm.

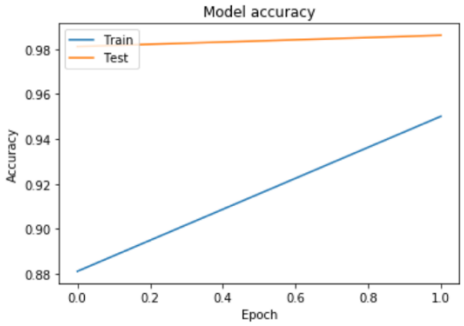
When batch size= 64, 64 samples from the training dataset will be used to find out the error gradient. This is done before the model weights are transformed. 1 training epoch means that the learning algorithm has made one pass through the training dataset.
Following results are obtained when we change Batch size in various configurations:
When the batch size is changed to 1024, more conv layers are also added along with dropout in the conv layer after each operation. The results are as follows:

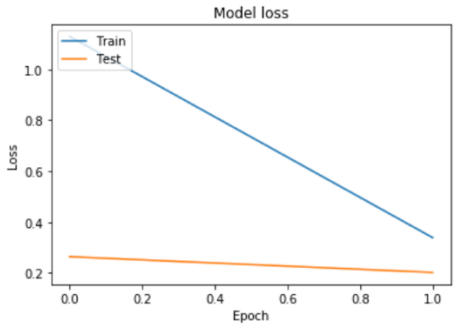
Batch Size	1024	64
Epoch	2	2
Accuracy (train)	93%	95.01%
Loss(train)	0.84	0.761

Some important things to consider:

- lower asymptotic test accuracy can be obtained with higher batch sizes
- lost test accuracy can be recovered from a larger batch size by increasing the learning rate
- If the number of sample's are similar, then larger batch sizes make larger gradient steps than the corresponding smaller batch sizes.

We notice a performance degrade when batch size is 1024 compared to 64 hence, we revert back to Batch size of 64 and tweak other parameters so as to get the best possible accuracy





Varying the Learning rate:

Learning rate is a hyper-parameter that controls the adjustment of weights of the network in context to loss gradient. If the value is lower, we travel slower along the downward slope. If learning rate is too high, gradient descent can overshoot the minimum. It may fail to converge, or even diverge. The learning rate has a huge impact on how quickly our model can converge to the respective local minima.
Learning rate in one of the most crucial hyperparameter that needs to be tweaked. This is because it directly corresponds to how quick and efficiently the corresponding model can adapt to our classification job. If we equip smaller learning rate, we might need a greater number of training Epoch's. This is since we any type of changes are made the weights get updated and as a result if our learning rate is large enough, we can move across it in fast succession thereby utilizing fewer number of epoch's during the training phase. On the other hand, if the learning rate that we are utilizing for our neural architecture is too small then this can further contribute to the process being stuck.
However, a larger learning rate always ensures that the resulting model can converge quickly. A learning rate that is too small can lead to the process being stuck while on the other hand a learning rate that is too big can cause the model to converge too quickly. The challenge of training neural networks involves carefully selecting the learning rate. Higher learning rate yields a better result, although marginally.

Changing the number of Convolutional Layer and number of filters:

Each layer in a CNN has a wider scope of looking at different things. The first layer does not add a wide variety of things to look for because it is general. For example, as we keep moving deeper into the pipeline of a multi-layer network, we start encountering a variety of things that we would be looking for.

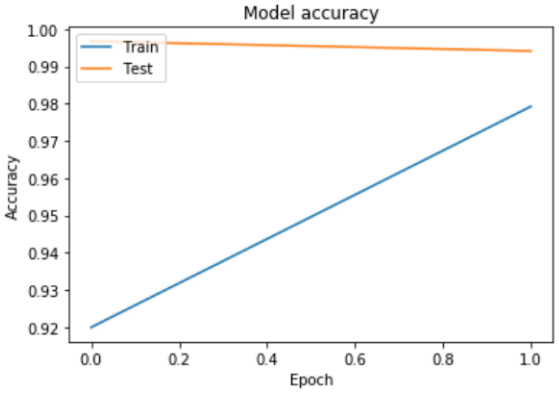
Some goes the number of filters; a larger value can be helpful to find out difficult features compared to the simpler features in the initial layers. Hence a large number of filters is always beneficial; however, there should be a threshold with respect to memory usage constraints and computation bandwidth.

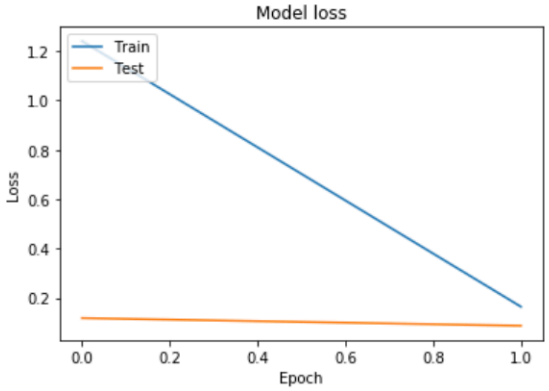
We add additional convolutional layers such that we have 7 conv layers along with added filters in each layer. Our batch size =64 and # of epochs=2

1. Number of filters for the first 3 conv2D layers: 128-128-128
2. Number of filters for the next 3 conv2D layers: 256-256-256
3. Number of filters for the last 1 conv2D layers: 512

	Last config	New layer Config (128x3| 256x3|512)
Epoch	Epoch=2	Epoch=2
Accuracy (train)	93%	97.9%
Loss(train)	0.84	0.16

```
Epoch 1/2
937/937 [==============================] - 76s 81ms/step - loss: 1.2
394 - accuracy: 0.9200 - val_loss: 0.1184 - val_accuracy: 0.9967
Epoch 2/2
937/937 [==============================] - 74s 79ms/step - loss: 0.1
646 - accuracy: 0.9792 - val_loss: 0.0873 - val_accuracy: 0.9941
```





Varying the training set size.

When we change the training set size it has a significant performance effect on our system. In practice more training samples always lead to a better accuracy for both train as well as test cases. But this isn't guaranteed every single time.

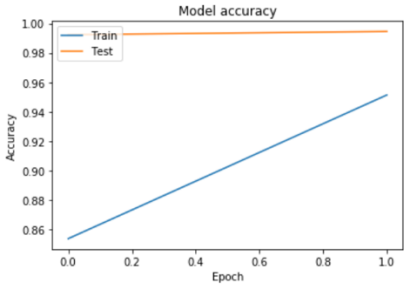
Thus, it is evident that since we have a higher training data in case where we split the data in training and validation into equal half. The model also is lower in case of more training data. This is not always the case as lots of training data may lead to the model performing better on the train set but not necessarily on the test. Thus, it can be concluded that we validation set reduces i.e. more training set hence it likely that we will get a better accuracy both in terms of test as well as train.

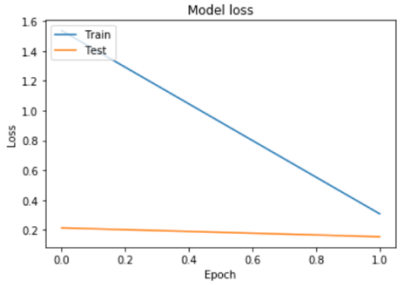
Changing the kernel size for the Convolution

Kernel first has the power to reduce the input size dimensions if no padding is used. Moreover, it's size matters with the assumptions of the size of the local features in the input. Also, the learning parameters are associated with the kernel. Bigger the size, means more learnable parameters. Hence, there is no benefit attach alone with smaller or bigger kernel. We always need to play with kernel size along with the number of features, and the number of layers.

Changing the kernel size to 2 throughout we get following results:

```
Epoch 1/2
937/937 [==============================] - 68s 72ms/step - loss: 1.5
354 - accuracy: 0.8540 - val_loss: 0.2143 - val_accuracy: 0.9923
Epoch 2/2
937/937 [==============================] - 64s 68ms/step - loss: 0.3
088 - accuracy: 0.9514 - val_loss: 0.1551 - val_accuracy: 0.9945
```





The training values seem to take a bit of hit downwards hence we default back to the old kernel size

Changing activation functions in the output layer:

The main task of the activation function is to put non- linearity relation into the output of a neuron. It decides where a neuron should be activated or not. Activation function is an important parameter since it decides, whether a neuron should be fired or not This is done by computing weighted addition and summing bias to it. The prime motive of the activation function is to introduce non-linearity into the output of any neural network architecture.

The main reason behind using different activation functions is to analyze what is its effect on prediction and performance metrics. sigmoid, hyperbolic tangent Softmax and ReLU are some of the choices available. Each function has different characteristics which make them interesting to explore. Sigmoid function has a well-defined non-zero derivative everywhere, allowing gradient descent to make some progress at every step. Tanh function varies smoothly between [-1, 1] whereas ReLU remains dormant till [0, 0] and then varies linearly.

RELU: Mainly used in hidden layers of Neural network. RELU is less computationally expensive than tanh and sigmoid since it focusses on simpler mathematical operations. Only a few neurons are activated making the network sparse making it efficient and easy for computation.
RELU has an advantage that it can learn much quicker compared to other activation functions. it is a better idea to use SoftMax at the output layer such that we get classification as a probabilistic distribution because we are concerned with multi classification

Sigmoid: The range of Sigmoid function is between [0,1] and the result can be calculated to be 1 if value > 0.5 and 0 if value< 0.5. Sigmoid happens to be a better choice especially in cases where binary classification is concerned

Tanh: Helpful in hidden layers of a neural network as its values lies in [-1 to 1] and the average/mean for hidden layer is almost 0 or very close to it, as a result it helps in positioning the data at the center by bringing mean close to 0. As a result, the learning for the next layer is much easier.

Advantages of using ReLUs- reduced probability of vanishing gradient.
This is due to the fact that the gradient has a constant value in contrast, the gradient of sigmoid becomes increasingly small as the absolute value of x increases. The constant gradient of ReLUs results in faster learning.

Advantages of using Softmax: At the end of a network, we can either use nothing (logits) and get a multi parameter regression or a sigmoid and get a number between 0 and 1 for each output, this is useful when we have several possible and independent answers or use SoftMax where all the outputs sum to 1, this useful when we want one best answer, we get a probability distribution. In one to one comparison if we are well aware of our function, Softmax seems to be a better choice although a lot of other parameters also contribute to a great extent.

Additionally, Relu/SoftMax is in general a better choice in the hidden layer since it is much faster computationally as an activation function. Thus, Softmax is the best solution at the output layer since we want to classify images. RELU/Softmax learns much faster than sigmoid and Tanh function. Hence, we observe that SoftMax got the highest accuracy

‘RELU’ is used after each convolution since we need to apply elemt wise activation function on the convolved image. Activation function produces an output if we have enough inputs to other layers.

Changing the optimizer:

Theoretically Adam and Adelta should be better than SGD since it computes individual learning outcome rates for different parameters. But the performance does depend a lot on other parameters too. In this implementation we change the optimizer from Adelta to SGD (Stochastic gradient descent) and Adam and observe the necessary changes. However, it is quite one dimensional to compare optimizers directly without knowing other parameters such as momentum for example which can contribute to the model performance.

Changing the Dropout after Convolution:

Dropout is essential to reduce overfitting i.e. when our model does better on the training data but worse on the test. However, it is important to understand where to implement dropout i.e. in the fully connected layer vs at the convolutional layers. **It is always better to use dropout in Fully connected layers vs at the convolutional layer due to the following reasons:** Convolutional layers usually don't have all that many parameters, so they need less regularization to begin with. Another is that, because the gradients are averaged over the spatial extent of the feature maps, dropout becomes ineffective: there end up being many correlated terms in the averaged gradient, each with different dropout patterns. So, the net effect is that it only slows down training but doesn't prevent co-adaptation. One way to mitigate this is to ensure that you apply the same dropout mask in every spatial position within a layer per example, but this may end up being too strong of a regularizer.

In the several above results we already have dropout values added after each convolution except out base model

Effect of Batch Normalization:

Batch Normalization is another method to regularize a convolutional neural network. On top of a regularizing effect, batch normalization also gives your convolutional network a resistance to vanishing gradient during training. This can decrease training time and result in better performance.

Batch normalization is used along with dropout as a regularization technique. It is already implemented in the above implementations.

Changing the Momentum:

We use a momentum term in the objective function, which is a value between 0 and 1 that increases the size of the steps taken towards the minimum by trying to jump from a local minimum. If the momentum term is large, then the learning rate should be kept smaller. A large value of momentum also means that the convergence will happen fast. But if both the momentum and learning rate are kept at large values, then we might skip the minimum with a huge step. A small value of momentum cannot reliably avoid local minima and can also slow down the training of the system. Momentum also helps in smoothing out the variations, if the gradient keeps changing direction. A right value of momentum can be either learned by hit and trial or through cross-validation.

Changing number of neurons in the fully connected layer:

One of the most critical design parameters is deciding on the # of neurons in a hidden layer architecture of a typical neural network. These layers are termed hidden since we never have access to the weights present inside them, i.e. they represent an intermediate layer between the input layer and output layer. A large number of neurons in the hidden layer can drastically increase the time to train the network. More importantly the amount of training time can inflate to the point where it is not possible to properly train the neural network.

Underfitting occurs when there are too few neurons in the hidden layers to adequately detect the signals in a complicated data set.
Using too many neurons in the hidden layers can result in several problems. Thus, it can be said that Overfitting occurs when the neural network has so much information processing capacity that the limited amount of information contained in the training set is not enough to train all of the neurons in the hidden layers.

selection of number of neurons for any particular architecture will be better if we use trial and error out method. This is due to the fact that if we equip trial and error method with more neurons, we can obtain the desired highest accuracy.

The model loss vs Epoch's for 256-256-512-10 vs 512-512-1024-10 along **with Epoch=1**; Batch size =64; adding 7 conv layers and adding regularization after each conv.

Accuracy vs loss can be seen from the table below:

Model	Accuracy	Loss
256-256-512-10 neurons	93%	0.84
512-512-1024-10 neurons	95.9%	0.08

1. As evident from the table it is observed that, with 512-512-1024-10 neurons in the hidden layers we can see the highest accuracy. However, there is a limit to this, and hence too many neurons must be avoided to make sure that the test accuracy is sufficiently high.
2. Also, the loss lowers down by a little, although other parameters need to be tweaked to observe significant performance metric.

Changing the number of Epoch's (included in the best performing model)

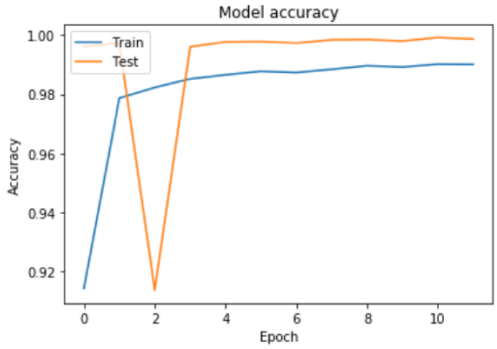
Epoch can be defined as a measure of the number of times the training vectors are utilized in order to update the corresponding weights. The numbers of epochs represent how different the data is. In our implementation following results are obtained: When the number of Epoch's are increased the accuracy increases by a lot for CNN especially where the training accuracy benefits are evident. This can be seen from below:

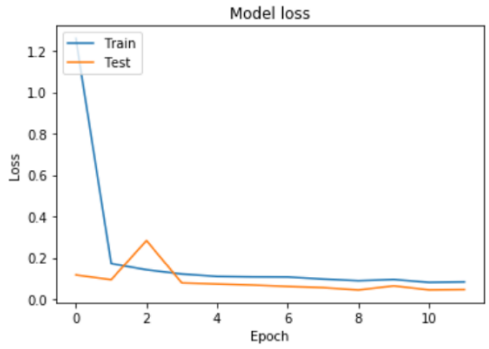
Model	Accuracy (training)	Loss(training)	Accuracy Valid	Loss valid
12 epochs				
1 epoch's	0.93	0.84	0.95	0.71

Thus, there is a direct correlation between the accuracy and loss when we tweak number of epochs. As the number of epochs increases, a greater number of times the weight is changed in the neural network and the curve goes from underfitting to optimal to overfitting curve. The numbers of epochs are related to how diverse your data is. When the entire dataset is passed during both forward as well as backward during Back propagation it is termed as 1 complete Epoch. Thus, it is a metric using which the number of times all of the training vectors are used once to update the weights. The numbers of epochs are related to how diverse the data is. one epoch is not enough, and hence we need to pass the dataset over multiple times to the same neural network so as to get a better accuracy.

The following are the results with all **previous configuration settings and epochs set to 12**

```
Epoch 11/12
937/937 [==============================] - 72s 77ms/step - loss: 0.0
808 - accuracy: 0.9902 - val_loss: 0.0445 - val_accuracy: 0.9992
Epoch 12/12
937/937 [==============================] - 72s 77ms/step - loss: 0.0
830 - accuracy: 0.9901 - val_loss: 0.0462 - val_accuracy: 0.9987
```





Changing loss functions:

- Neural networks require a loss function to calculate the model error as the training takes place with the help of an optimization process
- The loss function has an important task in that it must filter all parts of the model down into a single number in such a way that betterments in that number are a sign of a better performing model

categorical_crossentropy and mean squared are usually preferred in multi classification problems.

Conclusion:

Thus, various different parameters tweaks were performed on our CNN architecture and various results are obtained and performed
The best network after learning and tweaking all parameters can be explained as follows:

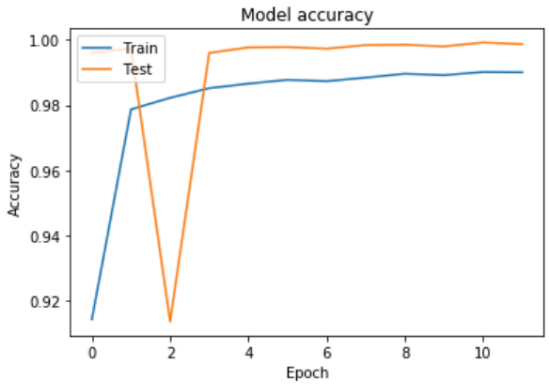
1. Number of filters for the first 3 conv2D layers: **128**
2. Number of filters for the next 3 conv2D layers: **256**
3. Number of filters for the last conv2D layers: **512**
4. number of neurons in the fully connected layer: **512-512-1024-10**
5. activation functions in the output layer: **SoftMax**
6. optimizer: **Adelta**
7. loss function: **Categorical Cross Entropy**
8. number of Convolutional Layer: **7**
9. Dropout after each Convolution: **0.3**
10. Batch Size: **64**
11. Momentum: **0.9**
12. kernel size for the Convolution: **3-3-5| 5-5-5|5**
13. Pool size for Max pooling: **2x2**
14. number of Epoch's: **12**

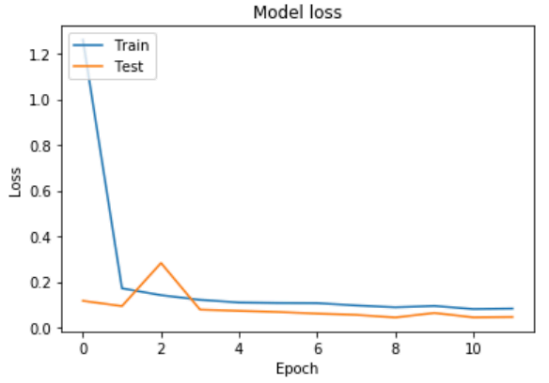
- We observed that number of neurons play a huge role, but it is also an important design criterion that too many neurons wouldn't contribute to an optimum design mechanism.
- To prevent overfitting dropout must be implemented. This can be done by adding dropout after each Convolutional layer.
- Batch size plays a critical role in the training accurate as it affects the learning rate of the model. Hence in this we prefer a batch size of 64 over 1024.
- Validation accuracy also gets a huge degradation after making changes to batch size and adding more convolutional layers. In theory however with more conv layers, large number of epoch's and dropout along with batch normalization should yield best performing model which is being calculated at epochs=10 along with other design tweaks.

Best model test Accuracy= 0.9901; Loss = 0.0462

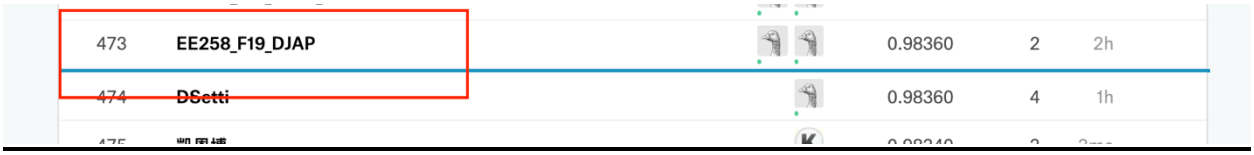
Validation Accuracy: 0.9987; Loss =0.0462

Best performing Model Accuracy and Loss Graph:





Kaggle Rank



Issues/ Things that I wanted to try:

- # of epoch's plays a critical role in CNN architecture but due to time constraints and Kaggle kernel automatically shutting off randomly I couldn't use their cloud-based GPU's and hence running everything locally took a whole lot time. Thus, I had to change multiple parameters at once instance since running for each with epoch>10 would be time consuming.
- With epochs >50 this model can get into top 100 Kaggle Ranking
- Play with some more parameters and multiple values. Although I tweaked around 15 values, I could only compare 2 values against each other for example in terms of epochs I did 2 and 12. Better things can be tried out such as cross comparisons between 5 types of optimizers batch size, learning rate etc. Same goes for other hyper parameters.