Deep Learning Project 1

Submitted by
Fatema Tabassum Liza
&
Arunima Mandal

Task-1

We have downloaded and pre-processed the train and test data according to the given link in the problem statement. Following on, We have designed three different neural networks according to the problem statement and the connection requirements. The overall connection and description of each neural network are described below along with their measurement metrics (accuracy and loss).

Part-1 (Fully - Connected)

The summary of this particular fully-connected neural network is shown below. Total 6 layers (4- hidden layers, 1-output layer and 1-input layer) have been used here.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_25 (Dense)             (None, 256)               65792
_________________________________________________________________
dense_26 (Dense)             (None, 128)               32896
_________________________________________________________________
dense_27 (Dense)             (None, 64)                8256
_________________________________________________________________
dense_28 (Dense)             (None, 32)                2080
_________________________________________________________________
dense_29 (Dense)             (None, 16)                528
_________________________________________________________________
dense_30 (Dense)             (None, 10)                170
=================================================================
Total params: 109,722
Trainable params: 109,722
Non-trainable params: 0
_________________________________________________________________
None
Train on 7291 samples, validate on 2007 samples
```

Fully-Connected network Test loss: 0.4534826671696993
Fully-Connected network Test accuracy: 0.9357249736785889

Part-2 (Locally Connected-unshared weights)

This type of layer is almost the same as the Convolutional layer but with only one important difference. In Locally-Connected Layer each neuron has its neuron and the filter is not shared among other neurons.

In Locally connected Layer the number of parameters are multiplied by the number of neurons. Hence, it is highly likely that the number of parameters increase drastically and if there is not enough data, the model might suffer from overfitting. However, this kind of network is beneficial for a few cases as this type of layer lets the network learn different types of features for different regions of the input.

The summary of the Locally-connected network that we have used is shown below.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
locally_connected1d_1 (Local (None, 254, 256)          260096
_________________________________________________________________
locally_connected1d_2 (Local (None, 252, 128)          24804864
_________________________________________________________________
locally_connected1d_3 (Local (None, 250, 64)           6160000
_________________________________________________________________
locally_connected1d_4 (Local (None, 248, 32)           1531648
_________________________________________________________________
locally_connected1d_5 (Local (None, 246, 16)           381792
_________________________________________________________________
flatten_1 (Flatten)          (None, 3936)              0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                39370
=================================================================
Total params: 33,177,770
Trainable params: 33,177,770
Non-trainable params: 0
_________________________________________________________________
None
Train on 7291 samples, validate on 2007 samples
```

Locally Connected Test loss: 0.3582614425497383
Locally Connected Test accuracy: 0.8973592519760132

Part-3 (Locally Connected-shared weights or Convolutional Neural Network)

Convolutional Neural Networks (CNNs) is a phenomenon in the field of image recognition. CNN is a deep learning algorithm which takes an image as an input and assigns weights and biases to various objects in the image which allows 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.

CNN is basically composed of a sequence of layers, and every layer of a ConvNet transforms propagates one volume of activations to another through a differentiable function. Mainly, three types of layers are used to build ConvNet architectures: Convolutional Layer, Pooling Layer, and Fully-Connected Layer. These layers are stacked on top of each other to form a full ConvNet architecture.

The summary of the Convolutional Neural network that we have used is shown below.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_11 (Conv2D)           (None, 14, 14, 32)        320
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 12, 12, 64)        18496
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
dropout_11 (Dropout)         (None, 6, 6, 64)          0
_________________________________________________________________
flatten_7 (Flatten)          (None, 2304)              0
_________________________________________________________________
dense_13 (Dense)             (None, 128)               295040
_________________________________________________________________
dropout_12 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_14 (Dense)             (None, 10)                1290
=================================================================
Total params: 315,146
Trainable params: 315,146
Non-trainable params: 0
_________________________________________________________________
None
Train on 7291 samples, validate on 2007 samples
```

CNN Model Test loss: 0.2874397947594548
CNN Model Test accuracy: 0.9686098694801331

Task-2

In this part of the project we have analyzed different aspects of three neural networks mentioned above and showed empirical results based on our experiment. The aspects of neural networks that we have studied are- Parameter initialization strategies for neural net, batch normalization strategies, how to vary learning rate and momentum in order to make our model more accurate and efficient.

Part 1 (Parameter Initialization Strategies)

Deterministic algorithms are great as they can make guarantees about best, worst, and average running time. The problem is, they are not suitable for all problems.

On the other hand, nondeterministic algorithms use elements of randomness when making decisions during the execution of the algorithm. This means that a different order of steps will be followed when the same algorithm is rerun on the same data. They can rapidly speed up the process of getting a solution, but the solution will be approximate, or "good," but often not the "best." Nondeterministic algorithms often cannot make strong guarantees about running time or the quality of the solution found. This is often fine as the problems are so hard that any good solution will often be satisfactory.

That does not mean that the nondeterministic algorithms are random; instead they make careful use of randomness. They are random within a bound and are referred to as stochastic algorithms.

Neural networks are trained using a stochastic optimization algorithm called stochastic gradient descent. The algorithm uses randomness in order to find a good enough set of weights for the

specific mapping function from inputs to outputs in your data that is being learned. It means that your specific network on your specific training data will fit a different network with a different model skill each time the training algorithm is run.

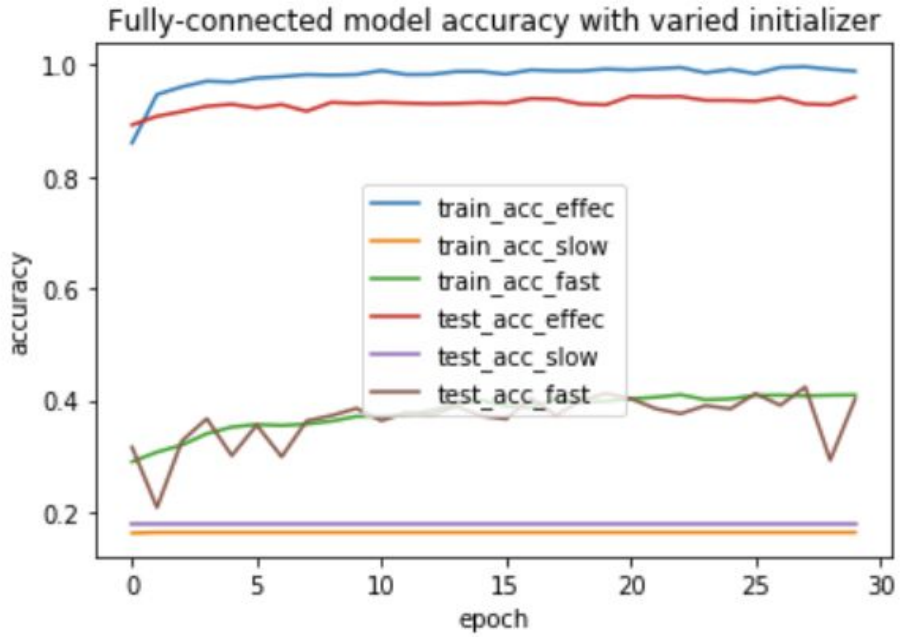
The weights of artificial neural networks must be initialized to small random numbers. The initialization of the weights of neural networks is a whole field of study as the careful initialization of the network can speed up the learning process. Modern deep learning libraries like Keras offer a host of network initialization methods, all are variations of initializing the weights with small random numbers. For example, some methods of Keras applicable for all network types are Zeroes, Ones, Constant, RandomNormal, RandomUniform, glorot_normal, glorot_uniform etc.
We have tried some of them in our 3 network models and found the following analysis:

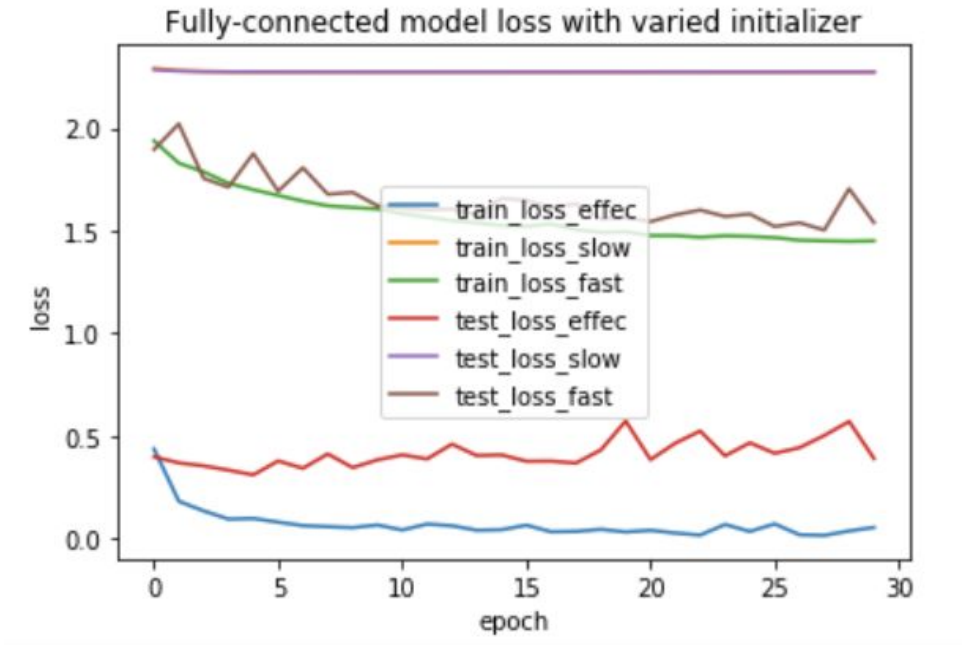
Fully-Connected Network:

We have analyzed several kernel initializers for the Fully Connected Network. Among them for 3 different kernel initializers, we have got 3 types of learning phase -

1. **Learning is very slow:** Initializer **Constant(value=100000)** was used. The train and test accuracy is the lowest in comparison to the other two cases. As the network's weights were initialized so far as 100000th position, it will take much greater time to converge. That's why the learning is very low for this initializer.
2. **Learning is effective:** Initializer **glorot_uniform** was used. The train and test accuracy is the highest in comparison to the other two cases.
   Glorot uniform initializer also called Xavier uniform initializer. It draws samples from a uniform distribution within [-limit, limit] where the limit is sqrt(6 / (fan_in + fan_out)) where fan_in is the number of input units in the weight tensor and fan_out is the number of output units in the weight tensor.
3. **Learning is too fast:** Initializer **Constant(value=0.01)** was used. The train and test accuracy shows medium performance in comparison to the other two cases. As the network's weights were initialized so close to 0, it will take much less time to converge.

That's why the learning is too fast for this initializer, resulting in moderate performance of the network.

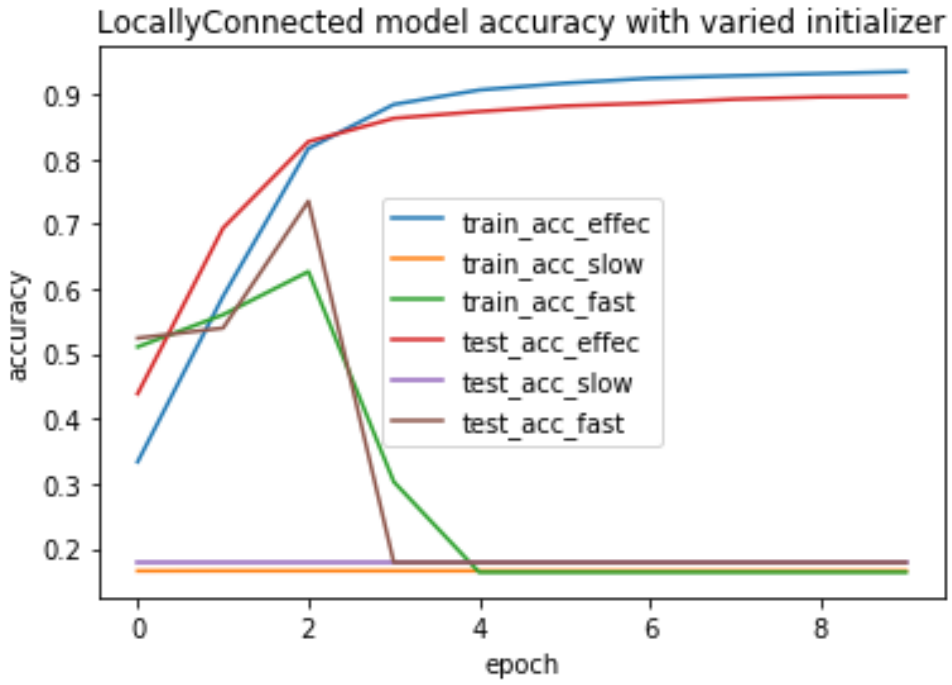


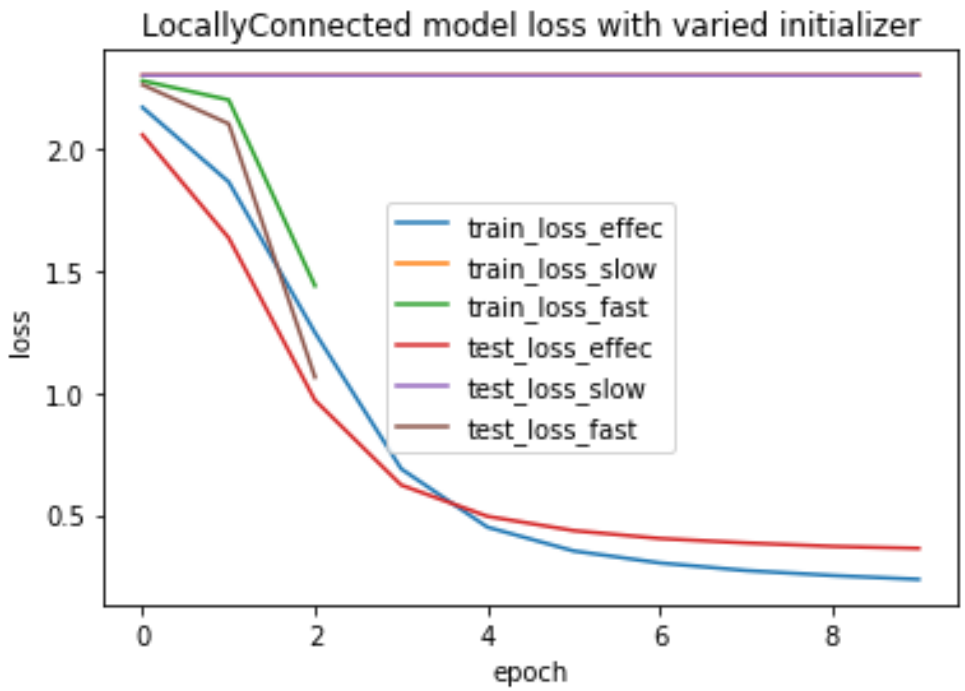


Locally-Connected Network:

We have analyzed several kernel initializers for the Locally-connected network. Among them, for 3 different kernel initializers, we have got 3 types of learning -

1. **<u>Learning is very slow:</u>** Initializer **Constant(value=100000)** was used. The train and test accuracy is the lowest in comparison to the other two cases. As the network's weights were initialized so far as 100000th position, it will take much greater time to converge. That's why the learning is very low for this initializer.
2. **<u>Learning is effective:</u>** Initializer **glorot_uniform** was used. The train and test accuracy is the highest in comparison to the other two cases.
3. **<u>Learning is too fast:</u>** Initializer **Constant(value=0.01)** was used. The train and test accuracy shows abrupt performance in comparison to the other two cases. As the network's weights were initialized so close to 0, it will take much less time to converge. Because the learning is too fast for this initializer, it shows abrupt behaviour.

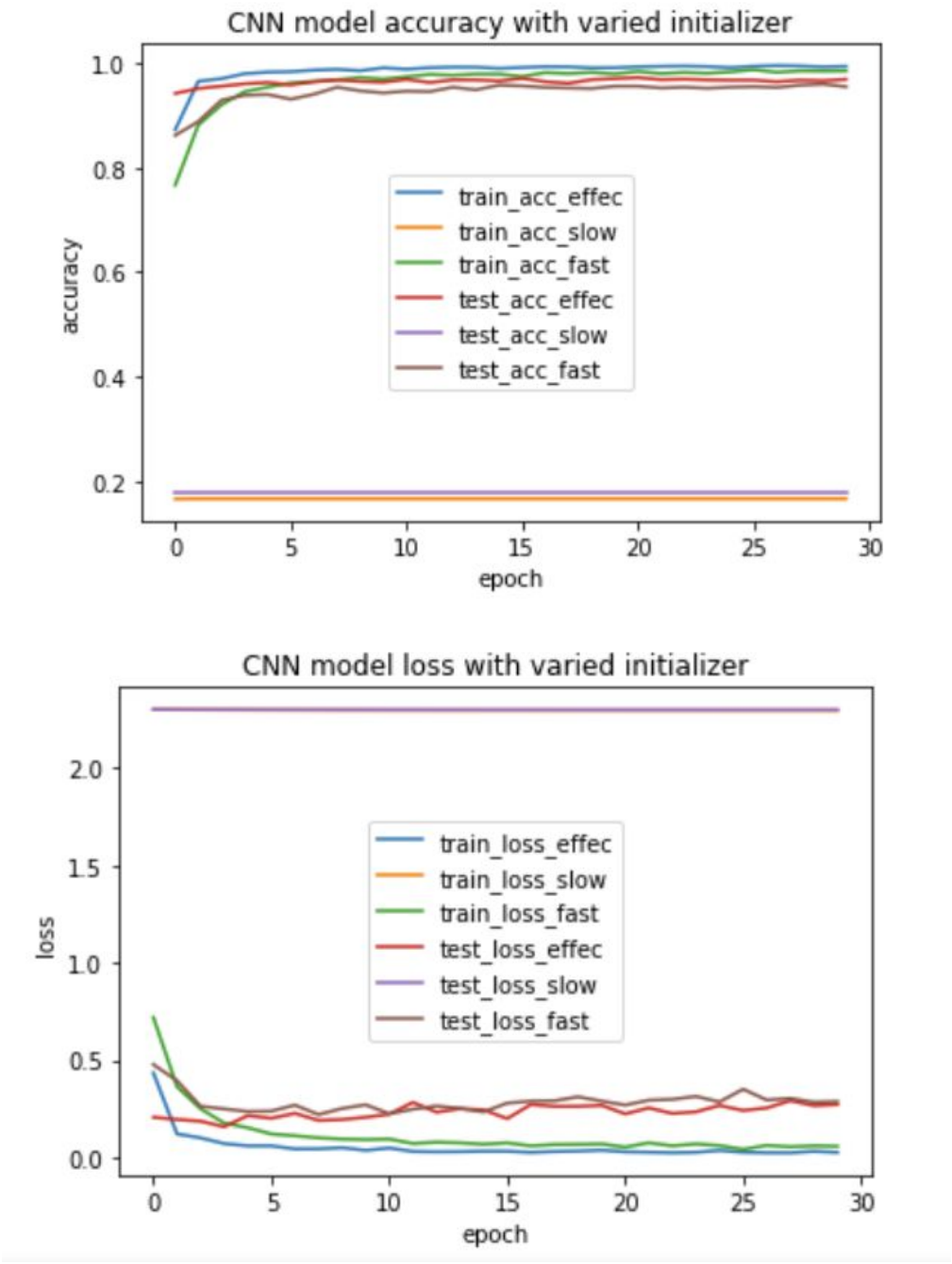




CNN :

We have analyzed several kernel initializers for CNN. Among them, for 3 different kernel initializers, we have got 3 types of learning -

4. **Learning is very slow:** Initializer **Constant(value=100000)** was used. The train and test accuracy is the lowest in comparison to the other two cases. As the network's weights were initialized so far as 100000th position, it will take much greater time to converge. That's why the learning is very low for this initializer.
5. **Learning is effective:** Initializer **glorot_uniform** was used. The train and test accuracy is the highest in comparison to the other two cases.
6. **Learning is too fast:** Initializer **Constant(value=0.01)** was used. The train and test accuracy shows medium performance in comparison to the other two cases. As the network's weights were initialized so close to 0, it will take much less time to converge. That's why the learning is too fast for this initializer, resulting in moderate performance of the network.



Part-2 (Learning Rate)

The challenge of training deep learning neural networks involves carefully selecting the learning rate. It may be the most important hyperparameter for the model.

In deep neural networks, Stochastic gradient descent is an optimization algorithm that estimates the error gradient for the current state of the model using examples from the training dataset, then updates the weights of the model using the back-propagation of errors algorithm, referred to as simply backpropagation.

The amount that the weights are updated during training is referred to as the step size or the **learning rate**. Specifically, the learning rate is a configurable hyperparameter used in the training of neural networks that has a small positive value, often in the range between 0.0 and 1.0.

The learning rate controls how quickly the model is adapted to the problem. Smaller learning rates require more training epochs given the smaller changes made to the weights each update, whereas larger learning rates result in rapid changes and require fewer training epochs. A learning rate that is too large can cause the model to converge too quickly to a suboptimal solution, whereas a learning rate that is too small can produce good results discovering the global minimum but can cause the process to get stuck.

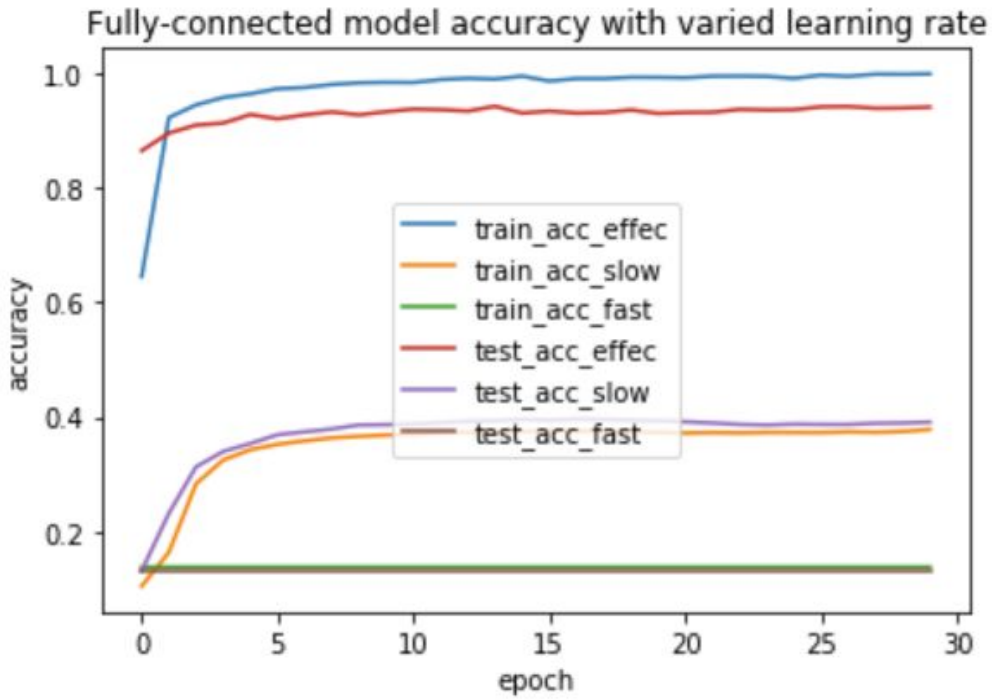
We have analysed several learning rate in our 3 network models and found the following analysis:

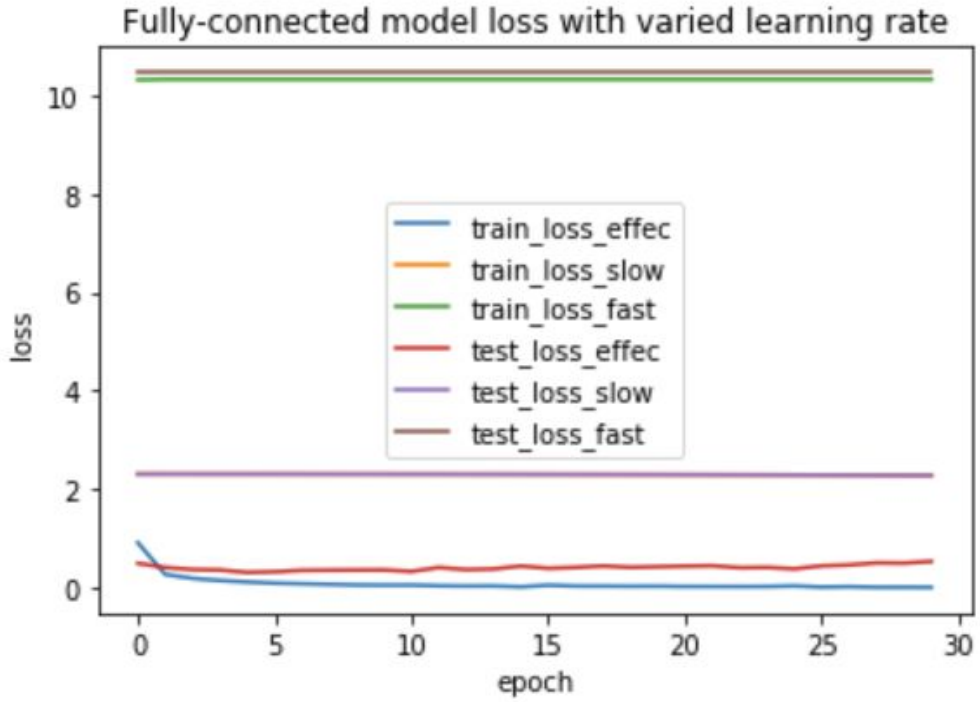
Fully-Connected Network:

We have analyzed some learning rates for the Fully Connected Network. Among them for 3 different learning rate, we have got 3 types of learning phase -

1. **Learning is very slow:** Learning rate = **0.000001** was used. The train and test accuracy is moderate in comparison to the other two cases. As the learning rate is too small, that means it will take much greater time to converge. Though it can find the global minimum which is a positive point, it may get stuck or might not be effective in overall for the network.
2. **Learning is effective:** Learning rate = **0.001** was used. The train and test accuracy is the highest here in comparison to the other two cases. This is one of the most common learning rates for many networks.

3. **<u>Learning is too fast:</u>** Learning rate = **50** was used. The train and test accuracy shows the lowest accuracy in comparison to the other two cases. As the learning rate here is too big i.e. the steps are big, it will take much less time to converge reaching some local optimum instead of the global optimum.



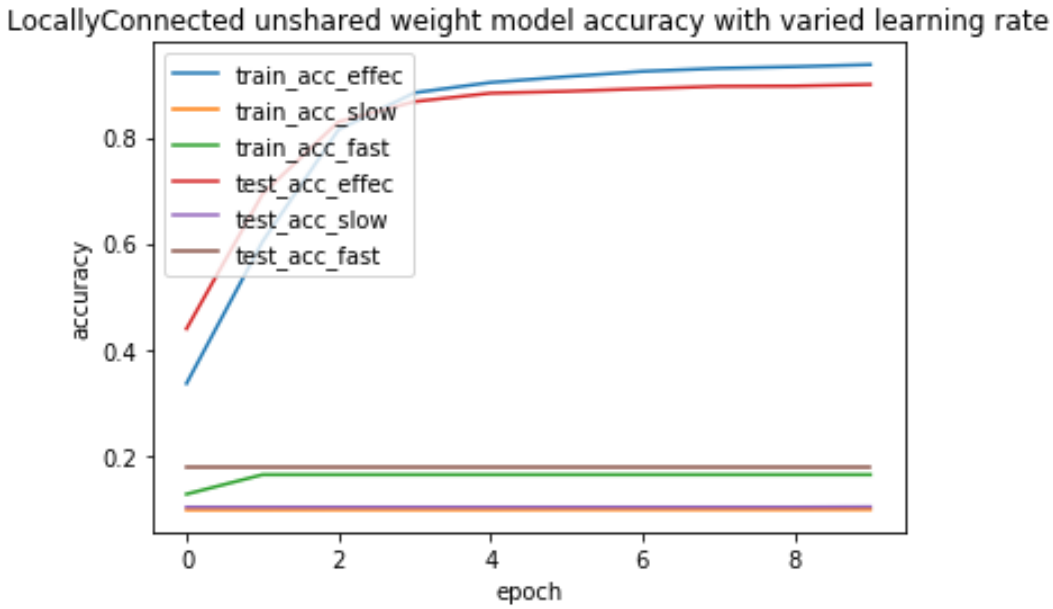


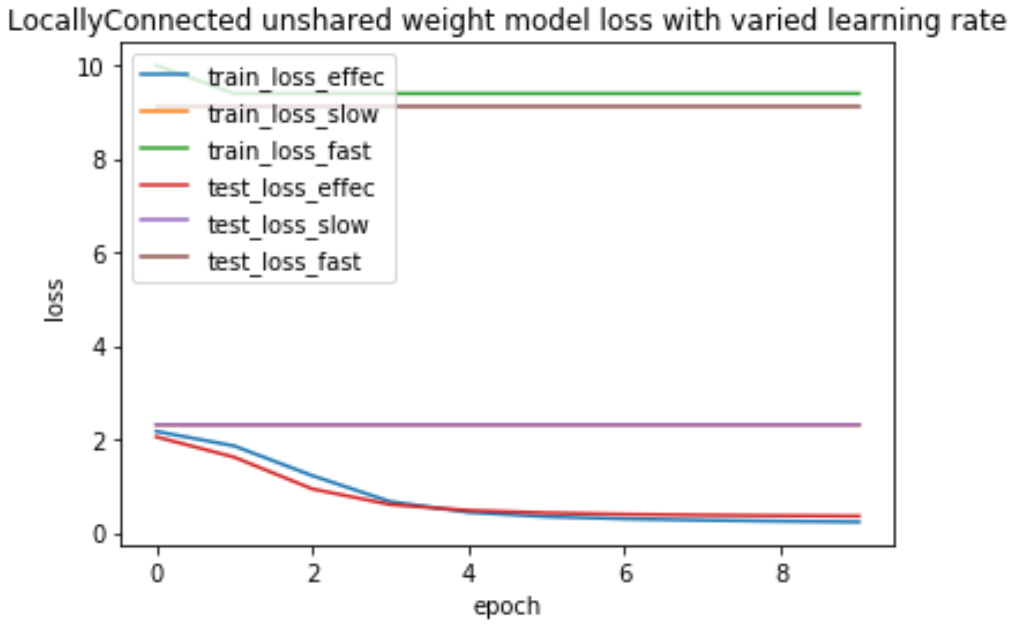
Locally-Connected Network:

We have analyzed some learning rates for the Locally Connected Network. Among them for 3 different learning rate, we have got 3 types of learning phase -

4. **<u>Learning is very slow:</u>** Learning rate = **0.000001** was used.Here, unexpectedly, the train and test accuracy is the lowest in comparison to the other two cases. As the learning rate is too small, that means it will take much greater time to converge, but it eventually must approach towards the global optimum value. We could not explain this abnormal behaviour of /this network.
5. **<u>Learning is effective</u>:** Learning rate = **0.001** was used. The train and test accuracy is the highest here in comparison to the other two cases. This is one of the most common learning rates for many networks.

6. **<u>Learning is too fast:</u>** Learning rate = **50** was used. The train and test accuracy is showing moderate performance in comparison to the other two cases. But we think it's not usual too as it is converging fast due to the greater learning rate, it should have the least accuracy or abrupt behaviour.

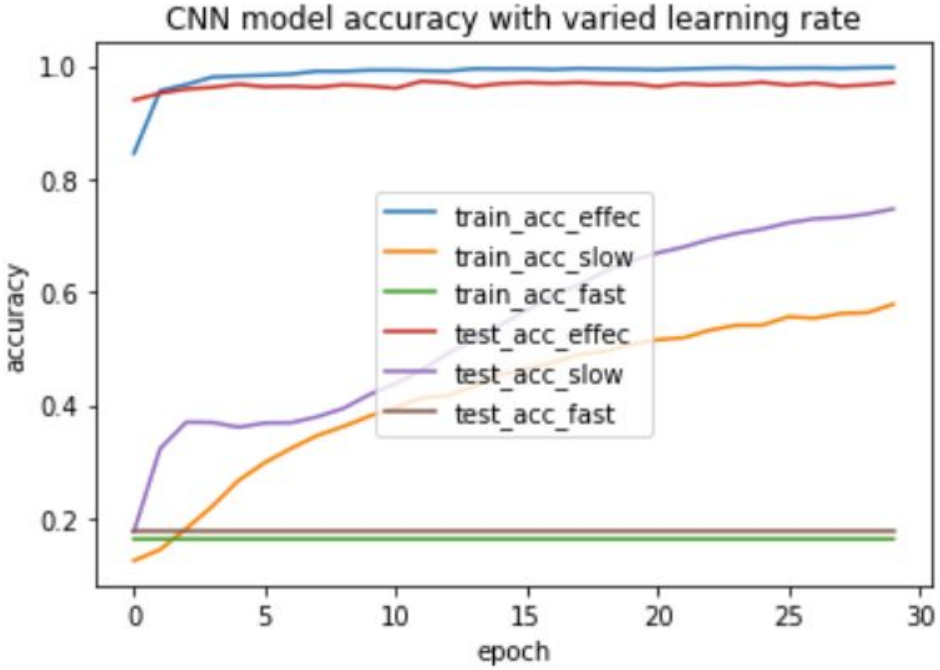


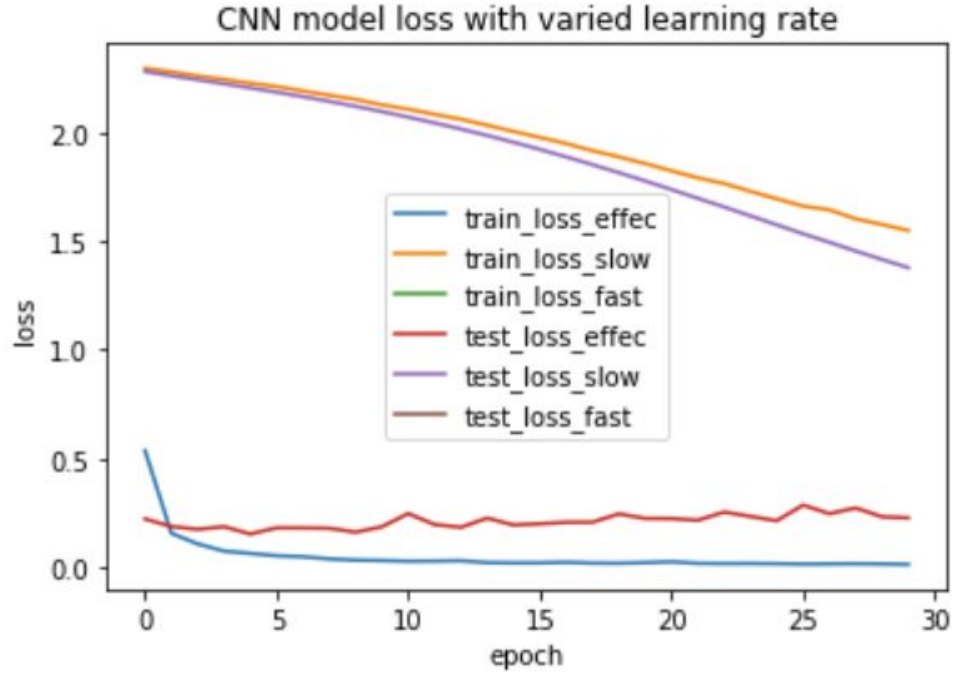


CNN:

We have analyzed some learning rates for CNN. Among them for 3 different learning rate, we have got 3 types of learning phase -

1. **Learning is very slow:** Learning rate = **0.000001** was used. The train and test accuracy is moderate in comparison to the other two cases. As the learning rate is too small, that means it will take much greater time to converge. Though it can find the global minimum which is a positive point, it may get stuck or might not be effective in overall for the network.
2. **Learning is effective:** Learning rate = **0.001** was used. The train and test accuracy is the highest here in comparison to the other two cases. This is one of the most common learning rates for many networks.
3. **Learning is too fast:** Learning rate = **50** was used. The train and test accuracy shows the lowest accuracy in comparison to the other two cases. As the learning rate here is too big i.e. the steps are big, it will take much less time to converge reaching some local optimum instead of the global optimum.


CNN model accuracy with varied learning rate
accuracy
epoch
train_acc_effec
train_acc_slow
train_acc_fast
test_acc_effec
test_acc_slow
test_acc_fast


CNN model loss with varied learning rate
loss
epoch
train_loss_effec
train_loss_slow
train_loss_fast
test_loss_effec
test_loss_slow
test_loss_fast

Part-3 (Batch size)

Neural networks are trained using gradient descent where the estimate of the error used to update the weights is calculated based on a subset of the training dataset.

The number of training examples used in the estimate of the error gradient is a hyperparameter for the learning algorithm called the “batch size,” or simply the “batch”. A batch size of 32 means that 32 samples from the training dataset will be used to estimate the error gradient before the model weights are updated. One training epoch means that the learning algorithm has made one pass through the training dataset, where examples were separated into randomly selected “batch size” groups.

The batch size impacts how quickly a model learns and the stability of the learning process. It is an important hyperparameter that should be well understood and tuned by the deep learning practitioner.

Batch normalization is a technique for improving the speed, performance, and stability of artificial neural networks [Reference: Wikipedia]. It is used to normalize the input layer by adjusting and scaling the activations. To increase the stability of a neural network, batch normalization normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation.

There is a direct impact of increasing the batch size, it can make the batch normalization stats (mean, variance) closer to the real population, and can also make gradient estimates closer to the gradients computed over the whole population allowing the training to be more stable (less stochastic), it should be noted that, there is a reason why we don't use the biggest batch sizes we can computationally afford.

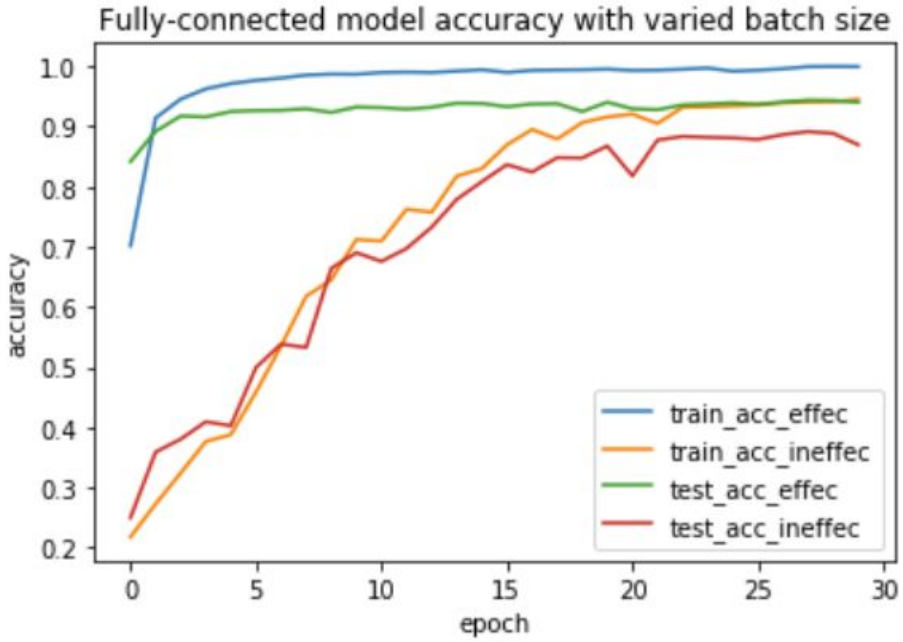
Let's suppose that our hardware configuration allows us to train using batches of 20K samples - our gradients would be more accurate and closer to the real gradient. However, this is not necessarily good because mini-batch SGD is proven to work better when the batches aren't big. As batches get remarkably bigger, mini-batch SGD becomes more and more like its father gradient descent, and this ancient monster isn't good for non-convex optimization problems like deep neural networks for both computational reasons and local-minima-related reasons.

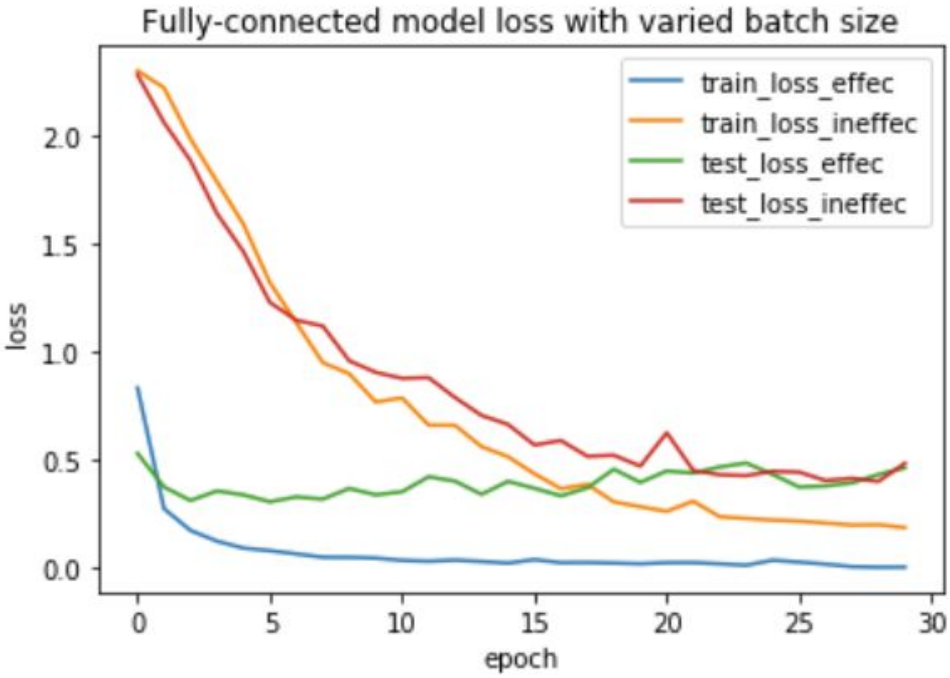
We have analysed several batch sizes in our 3 different network models and found the following analysis:

Fully-Connected Network:

We have analysed some batch sizes for the Fully Connected Network. Among them for 2 different batch sizes, we have got effective and ineffective batch size for the network models -

1. **Effective batch size:** We found batch size **16** as the effective batch size. We can see from the plotting that the train and test accuracy of the network increases rapidly for this batch size. That's because a small batch results generally in rapid learning but a volatile learning process with higher variance in the classification accuracy.
2. **Ineffective batch size:** We found batch size **1024** as the ineffective batch size. We can see from the plotting that the train and test accuracy increases very slowly in each epoch and ultimately acquire greater accuracy. That's because larger batch sizes slow down the learning process but the final stages result in a convergence to a more stable model exemplified by lower variance in classification accuracy.

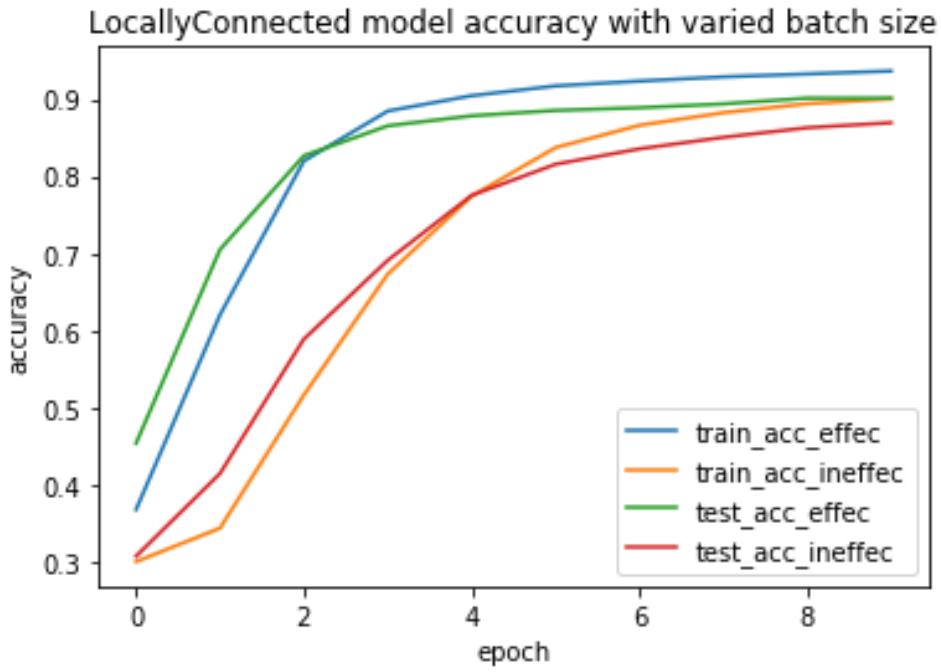


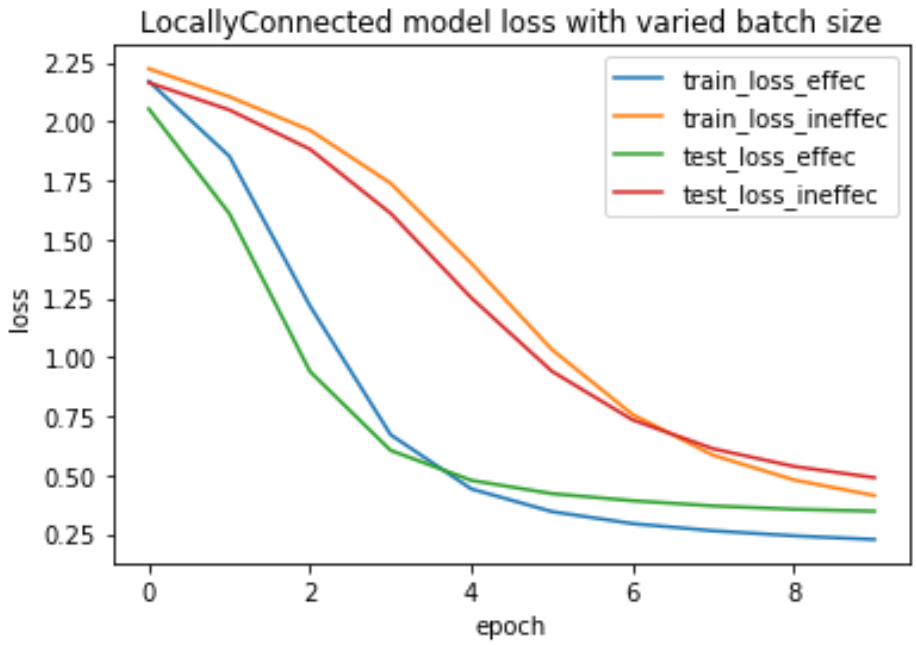


Locally-Connected Network:

We have analysed some batch sizes for the Locally Connected Network. Among them for 2 different batch sizes, we have got effective and ineffective batch size for the network models -

3. **Effective batch size:** We found batch size **16** as the effective batch size. We can see from the plotting that the train and test accuracy of the network increases rapidly for this batch size. That's because a small batch results generally in rapid learning but a volatile learning process with higher variance in the classification accuracy.
4. **Ineffective batch size:** We found batch size **32** as the ineffective batch size. We can see from the plotting that the train and test accuracy increases very slowly in each epoch and ultimately acquire greater accuracy. That's because larger batch sizes slow down the learning process but the final stages result in a convergence to a more stable model exemplified by lower variance in classification accuracy.

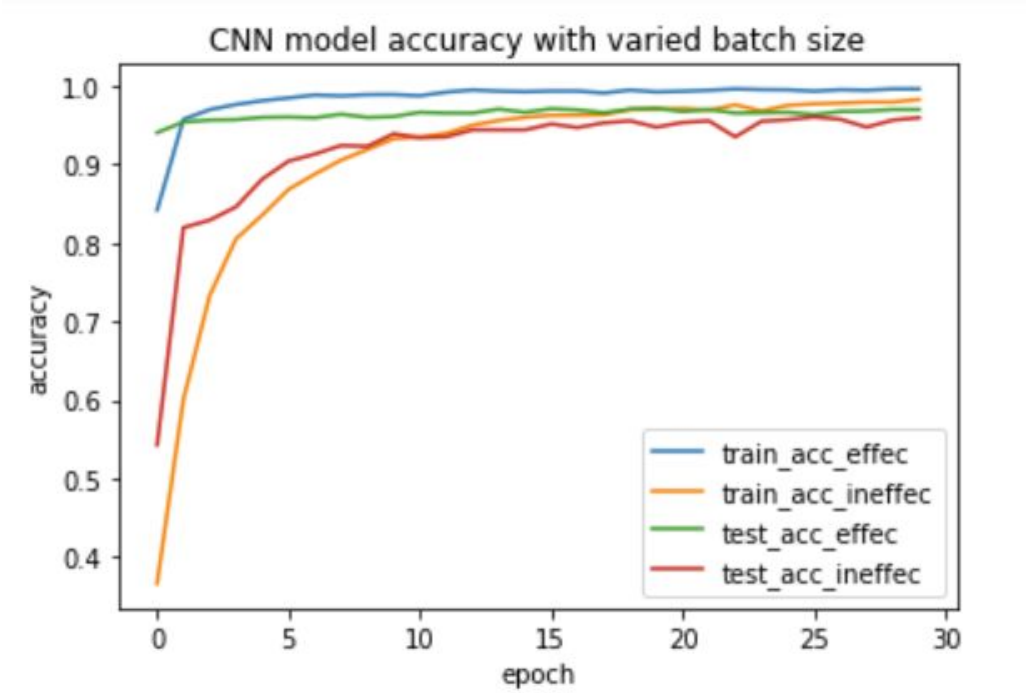


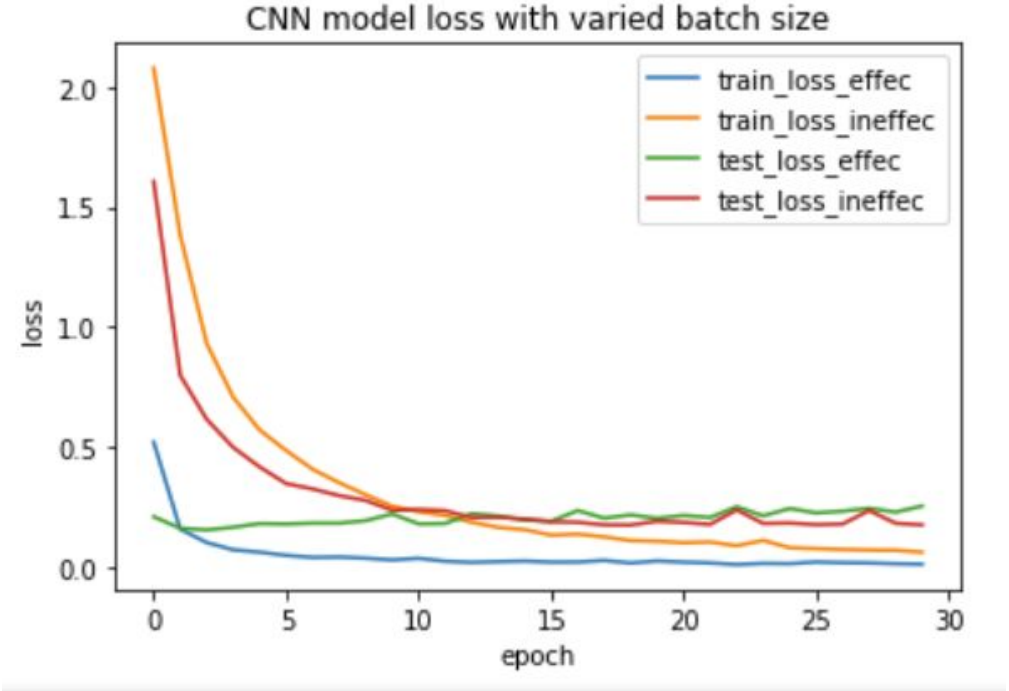


CNN:

We have analysed some batch sizes for CNN. Among them for 2 different batch sizes, we have got effective and ineffective batch size for the network models -

5. **<u>Effective batch size:</u>** We found batch size **16** as the effective batch size. We can see from the plotting that the train and test accuracy of the network increases rapidly for this batch size. That's because a small batch results generally in rapid learning but a volatile learning process with higher variance in the classification accuracy.
6. **<u>Ineffective batch size:</u>** We found batch size **1024** as the ineffective batch size. We can see from the plotting that the train and test accuracy increases very slowly in each epoch and ultimately acquire greater accuracy. That's because larger batch sizes slow down the learning process but the final stages result in a convergence to a more stable model exemplified by lower variance in classification accuracy.


CNN model accuracy with varied batch size
accuracy
epoch
train_acc_effec
train_acc_ineffec
test_acc_effec
test_acc_ineffec


CNN model loss with varied batch size
loss
epoch
train_loss_effec
train_loss_ineffec
test_loss_effec
test_loss_ineffec

Part-4 (Momentum)

A simple addition to classic SGD algorithm, called momentum, almost always works better and faster than Stochastic Gradient Descent. Momentum or SGD with momentum is a method which helps accelerate gradient vectors in the right directions, thus leading to faster converging. In simple words, momentum can smooth the progression of the learning algorithm that, in turn, can accelerate the training process. It is one of the most popular optimization algorithms and many state-of-the-art models are trained using it.

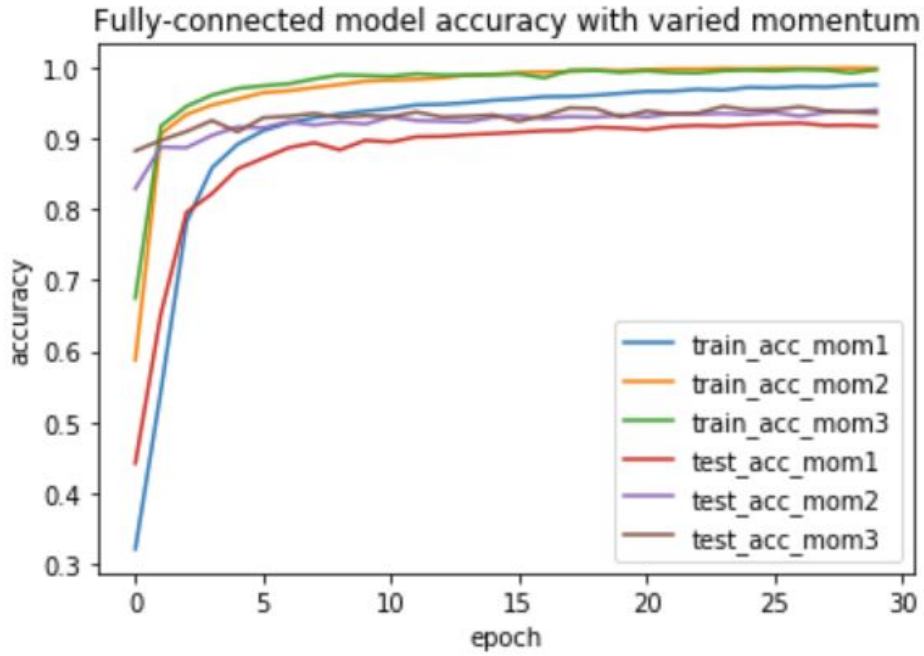
We were given the three momentum coefficient values - 0.5, 0.9, and 0.99 . Taking the best parameter initialization strategy, the best learning rate, and the best batch size we have found so far, we were asked to experiment with the three different momentum values on the three networks we have.

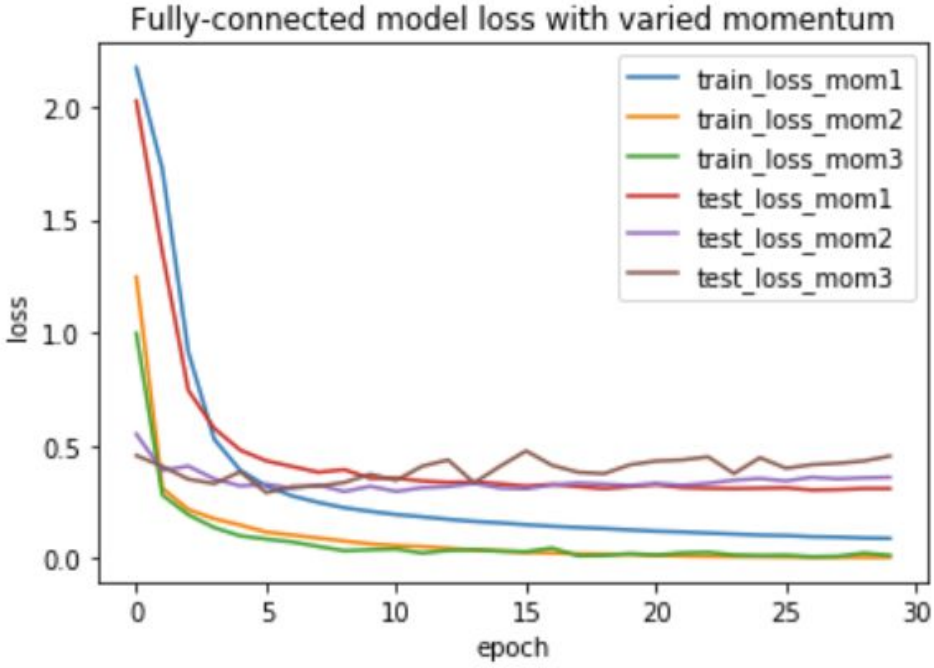
Fully-Connected Network:

For the Fully Connected Network we have, we have applied the below best parameters -

Best parameter initialization strategy: glorot_uniform
Best learning rate: 0.001
Best batch size: 16

We can see that the addition of momentum does accelerate the training of the model. Specifically, momentum values of 0.9 and 0.99 achieve reasonable train and test accuracy within about 10 training epochs. It is far better performance than when momentum is not used. In all cases where momentum is used, the accuracy of the model on the holdout test dataset appears to be more stable, showing less volatility over the training epochs.


Fully-connected model accuracy with varied momentum
accuracy
epoch
train_acc_mom1
train_acc_mom2
train_acc_mom3
test_acc_mom1
test_acc_mom2
test_acc_mom3


Fully-connected model loss with varied momentum
loss
epoch
train_loss_mom1
train_loss_mom2
train_loss_mom3
test_loss_mom1
test_loss_mom2
test_loss_mom3

Locally-Connected Network:

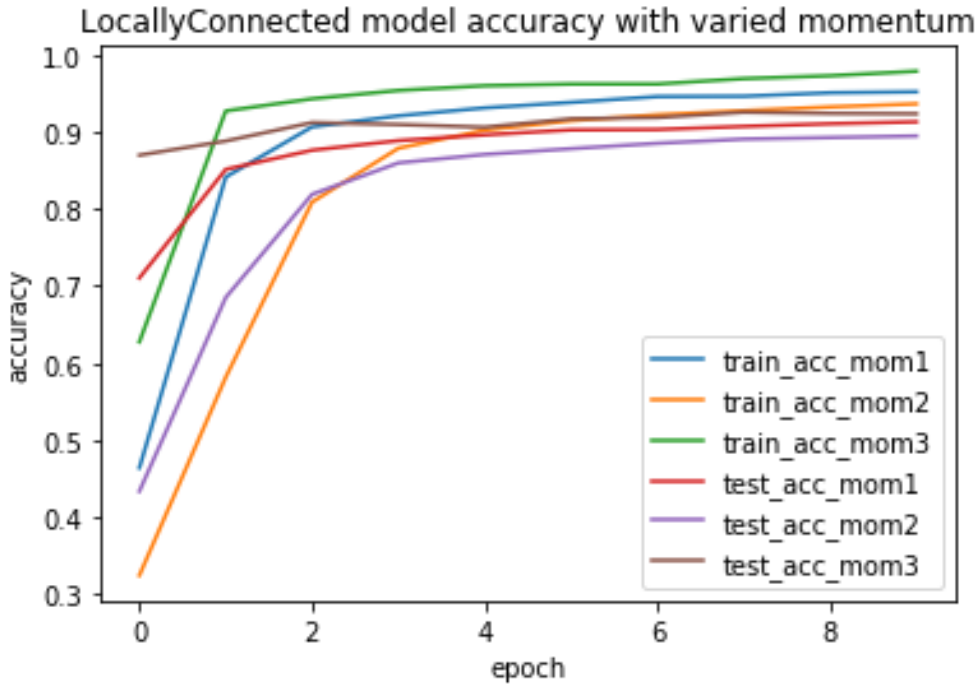
For the Locally Connected Network we have, we have applied the below best parameters -

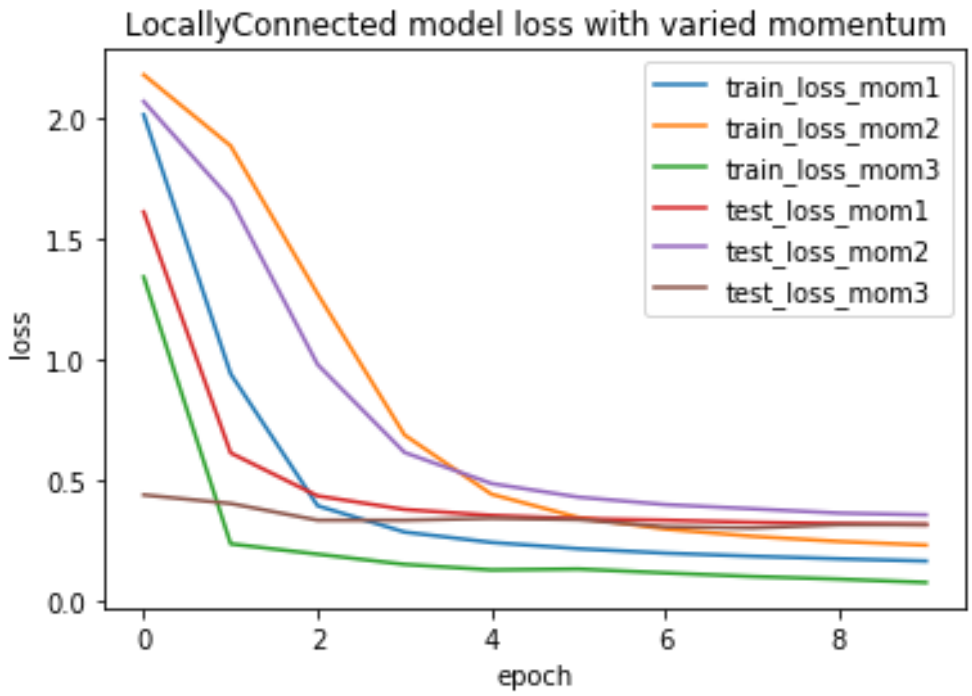
Best parameter initialization strategy: glorot_uniform

Best learning rate: 0.001

Best batch size: 16

We can see that the addition of momentum does accelerate the training of the model. Specifically, momentum values of 0.9 and 0.99 achieve reasonable train and test accuracy within about 10 training epochs. It is far better performance than when momentum is not used. In all cases where momentum is used, the accuracy of the model on the holdout test dataset appears to be more stable, showing less volatility over the training epochs.





CNN:

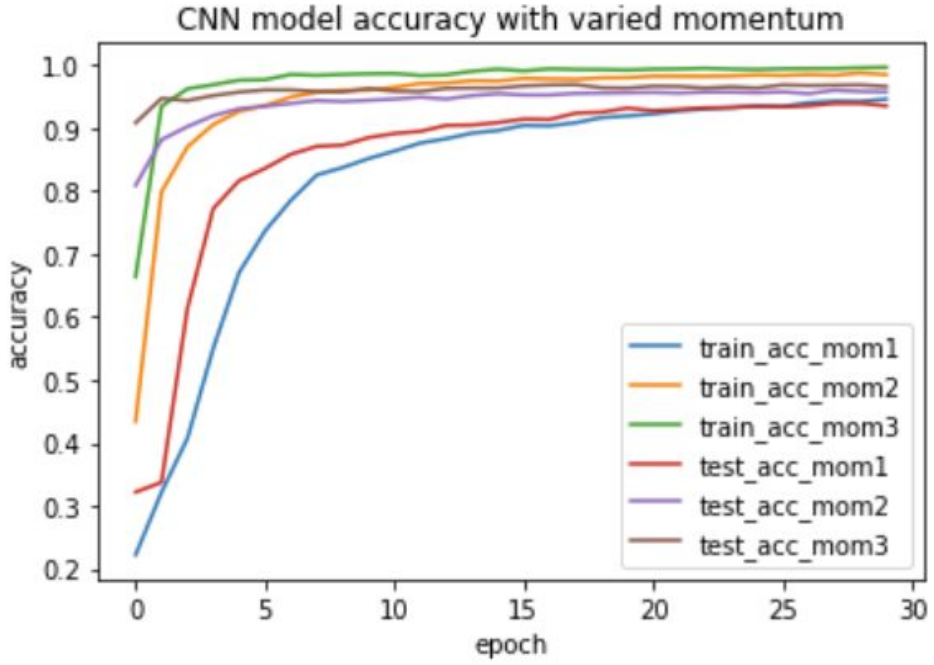
For CNN we have, we have applied the below best parameters -

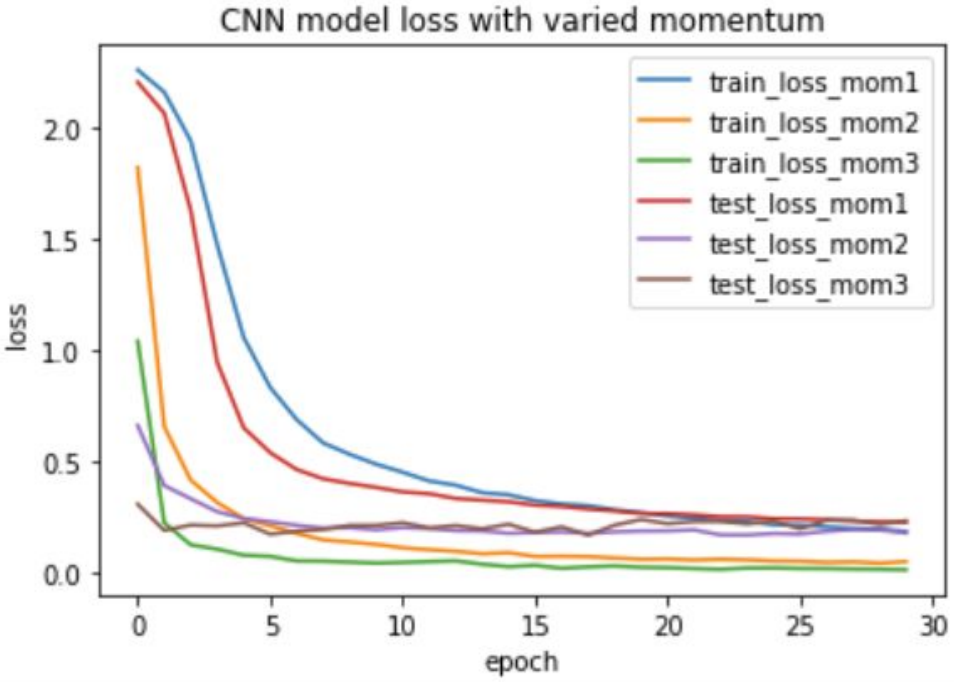
Best parameter initialization strategy: glorot_uniform

Best learning rate: 0.001

Best batch size: 16

We can see that the addition of momentum does accelerate the training of the model. Specifically, momentum values of 0.9 and 0.99 achieve reasonable train and test accuracy within about 10 training epochs. It is far better performance than when momentum is not used. In all cases where momentum is used, the accuracy of the model on the holdout test dataset appears to be more stable, showing less volatility over the training epochs.


CNN model accuracy with varied momentum
accuracy
epoch
train_acc_mom1
train_acc_mom2
train_acc_mom3
test_acc_mom1
test_acc_mom2
test_acc_mom3


CNN model loss with varied momentum
loss
epoch
train_loss_mom1
train_loss_mom2
train_loss_mom3
test_loss_mom1
test_loss_mom2
test_loss_mom3

Task 3

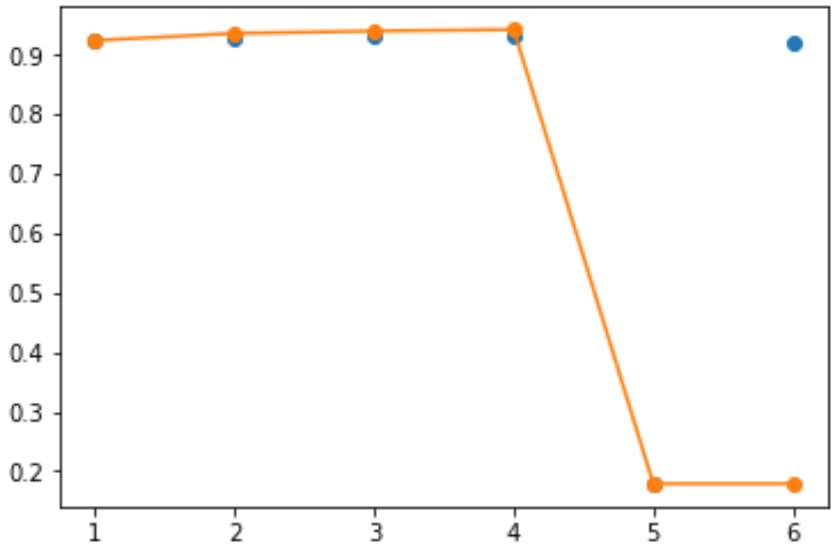
In this task we have studied how to improve Generalization. For this task, different types of regularization techniques have been used. One of the most common problem data science professionals face is to avoid overfitting which means that the model performs exceptionally well on train data but is not able to predict test data. Regularization is a widely used technique which makes slight modifications to the learning algorithm such that the model generalizes better

Part 1 (Ensemble)

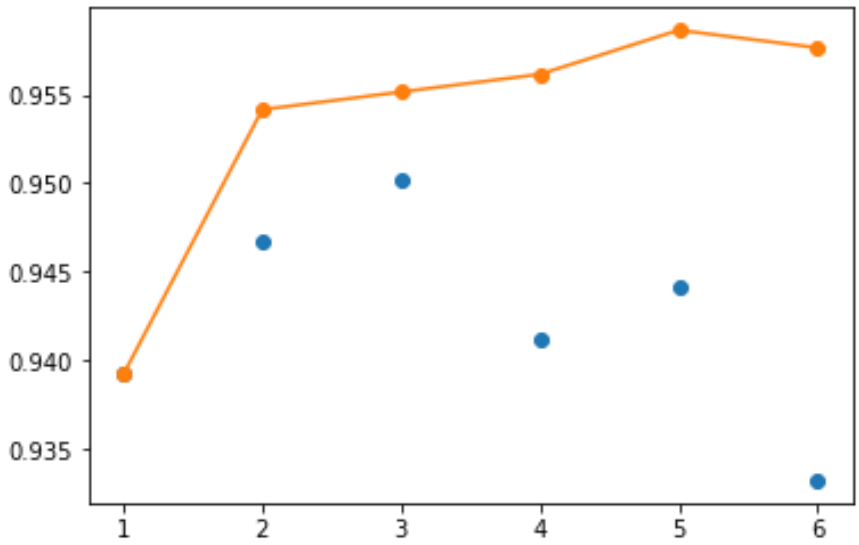
Ensemble learning are methods that combine the predictions from multiple models.
The models that comprise the ensambles should be good in making different prediction errors. Predictions that are good in different ways can result in a prediction that is both more stable and often better than the predictions of any individual member model. There are different methods to ensemble such as training each model on a different subset, or cross-validation and the bootstrap model. The models used in this estimation process can be combined in what is referred to as a resampling-based ensemble, such as a cross-validation ensemble or a bootstrap aggregation (or bagging) ensemble. In this part of the project we have chosen bootstrap aggregation (or bagging) ensemble and experimented with our train and test data to see ensemble vs single result.

We have used 6 additional neural networks with each of these specified networks (fully-connected, conv2D). In the bagging process usually the test error is considered for each of the additional neural networks to see how closely they comply with each other and then consider their vote for a particular class. In our process we calculate the average for all the networks for a particular class and then see which class has the highest value.

Our findings by using the ensemble method for Dense is shown below. X-axis holds the number of bagging ensemble models and y axis holds accuracy.



Our findings by using ensemble method for Conv2D are shown below. X-axis holds the number of bagging ensemble models and y axis holds accuracy.



The created line plot is encouraging. The blue data points indicate single model accuracy and the yellow indicate accuracy gained by bagging ensemble. We can see that the model accuracy increases where a single model performs poorly.

The difference between single model and ensemble model in the case of Dense and Conv2D is given below.

Estimated Accuracy dense 0.710 (standard-deviation=0.376
Dense > 1: single=0.926, ensemble=0.926
Dense > 2: single=0.924, ensemble=0.933
Dense > 3: single=0.928, ensemble=0.931
Dense > 4: single=0.933, ensemble=0.934
Dense > 5: single=0.179, ensemble=0.179
Dense > 6: single=0.917, ensemble=0.179
Estimated Accuracy conv 0.973 (standard-deviation=0.005)
Conv > 1: single=0.939, ensemble=0.939
Conv > 2: single=0.947, ensemble=0.954
Conv > 3: single=0.950, ensemble=0.955
Conv > 4: single=0.941, ensemble=0.956
Conv > 5: single=0.944, ensemble=0.959
Conv > 6: single=0.933, ensemble=0.958

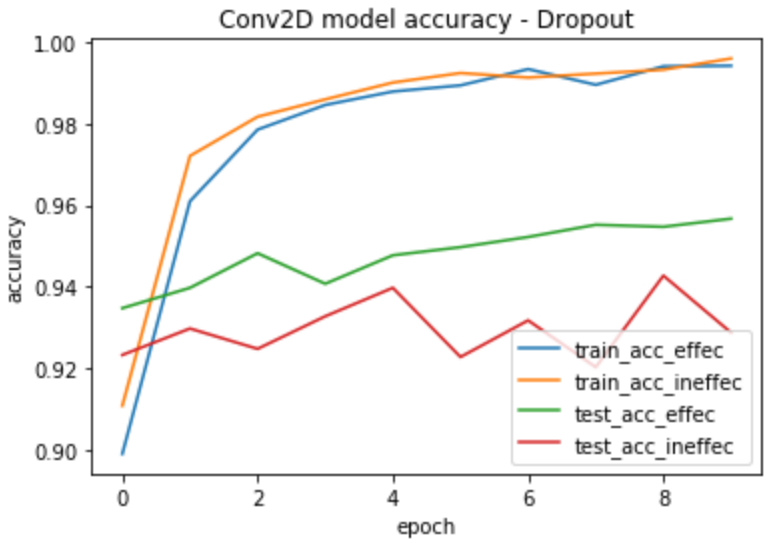
Part 2 (Dropout)

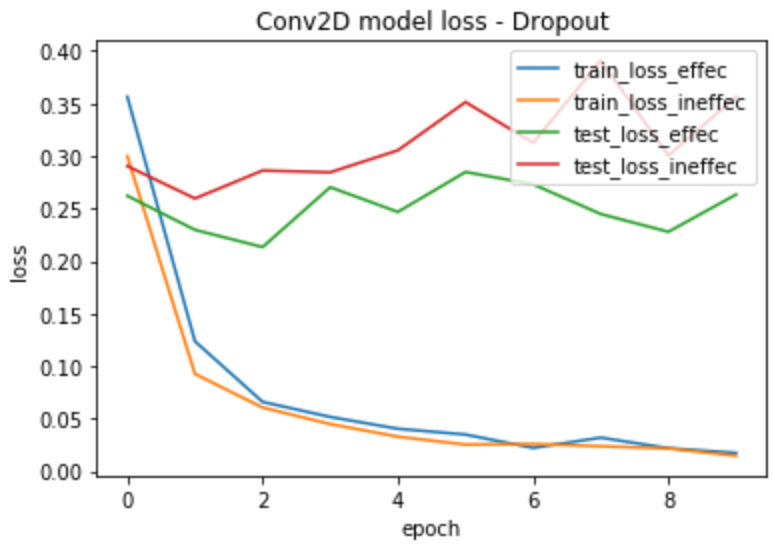
Dropout is a famous and powerful regularization technique. Dropout means that during training with some probability P a neuron of the neural network turned off during training. By using dropout networks it becomes simpler. A simpler version of the neural network results in less complexity that can reduce overfitting. The deactivation of neurons with a certain probability P

is applied at each forward propagation and weight update. This issue resolved the overfitting issue in large networks and suddenly more accurate Deep Learning architectures became possible.

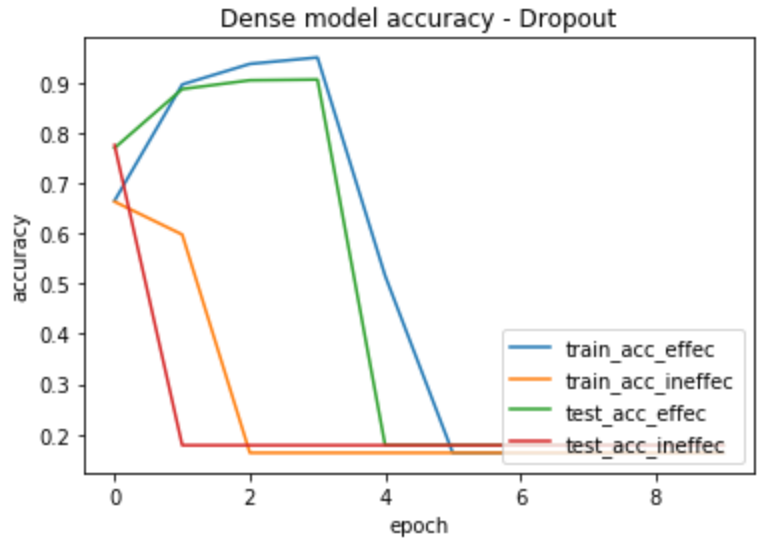
On each iteration, we randomly shut down some neurons (units) on each layer and don't use those neurons in both forward propagation and back-propagation. Since the units that will be dropped out on each iteration will be random, the learning algorithm will have no idea which neurons will be shut down on every iteration; therefore, force the learning algorithm to spread out the weights and not focus on some specific features (units).

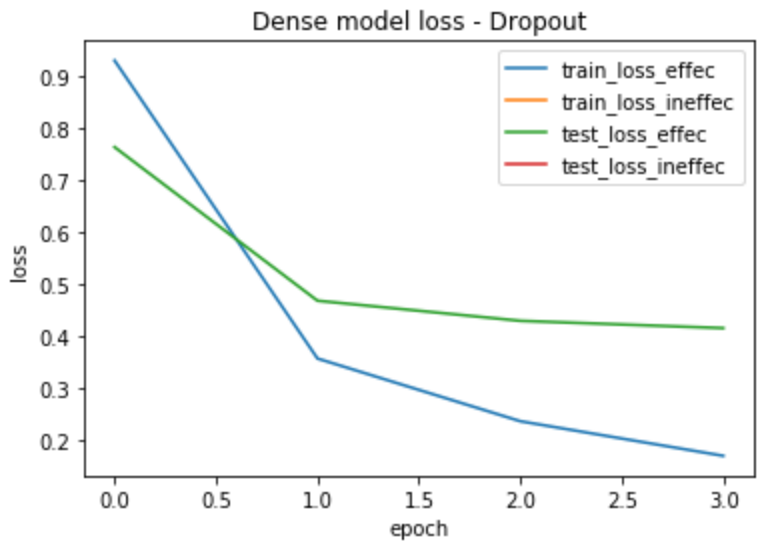
We have used dropout in our experimental neural net and saw the following result. We Have used varying dropouts and found out these results. For the effective result of conv2D we have added Dropout after the Dense layer.





We saw that dropout is ineffective or less efficient when applied right after convolutional layer. It results effectively when used in the Dense layer of convolutional neural network.





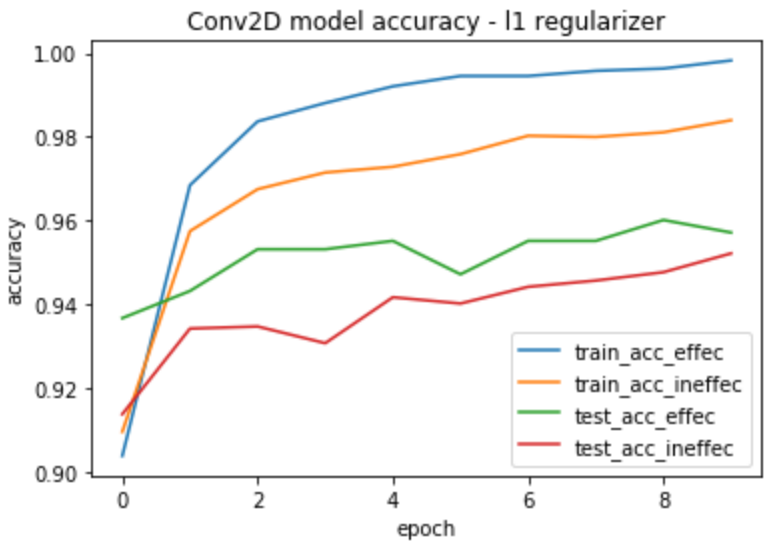
We can see that test loss is higher than train loss in dropout. However, the dropout model is not very effective in this model. We need to vary dropout values more to see which dropout values work the best for this model.

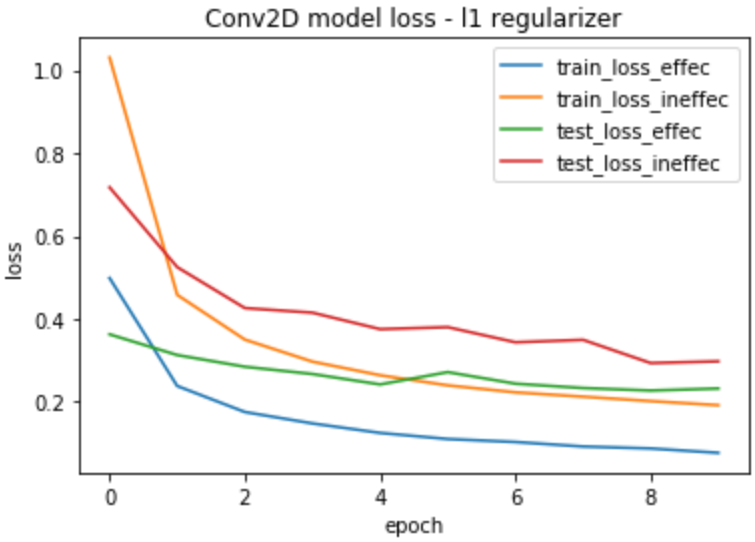
Part 2 (L1 Regularization)

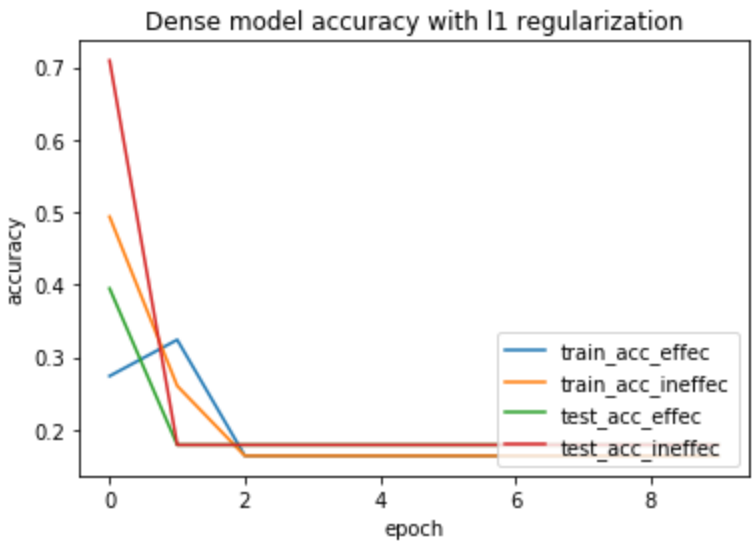
L1 regularization is another classic regularization technique. Norm regularization is a penalty imposed on the model's objective function for using weights that are too large. This is done by adding an extra term onto the function which is $\lambda|w|$ for L1. In these expressions, λ is a hyperparameter that controls the degree of regularization in the model. As in any classic regularization setup, adding this extra term will induce the model to balance the loss of its output against the magnitude of its weights.

L1 is generally better if the model uses certain inputs more heavily than others.

We have studied L1 regularization as well and analyzed the output for varying l1 value in activity_regularizer and plot graphs for some of the effective and ineffective cases. These are given below.


Conv2D model accuracy - l1 regularizer
accuracy
epoch
train_acc_effec
train_acc_ineffec
test_acc_effec
test_acc_ineffec


Conv2D model loss - l1 regularizer
loss
epoch
train_loss_effec
train_loss_ineffec
test_loss_effec
test_loss_ineffec



The Dense model accuracy using l1-regularization is not very good in this case. We can vary regularization with more l1 values or add l1 in varying layers to see which regularization values work best for this Dense model.

Task 4

Adversarial training

In many cases, Neural Networks seem to have achieved huma

In many cases, Neural Networks seem to have achieved human-level understanding of the task but to check if it really is able to perform at human-level, Networks are tested on adversarial examples. Adversarial examples can be defined as if for an input a near a data point x such that the model output is very different at a, then a is called an Adversarial example. Adversarial examples are intentionally constructed by using an optimization procedure and models have a nearly 100% error rate on these examples.

Adversarial training helps in regularization of models as when models when models are trained on the training sets that are augmented with Adversarial examples, it improves the generalization of the model.