Question 2: Classification: Convolutional Neural Networks

Data Preprocessing:
At first, we extracted the dataset. Then we converted the dataset into float 32 dtype. Scaling was done over the entire dataset by dividing it by 255 since the images were grayscale. This scaled data was used for all the algorithms.
We also visualized some of the unique items in the dataset.

```
#plotting some of the images of unique items
u, indices = np.unique(y_train_np, return_index=True)
plt.figure(figsize=(10,10))
for i in range(5):
    plt.subplot(1,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(X_train_sd[indices[i], :].reshape((28, 28)),cmap="gray")
    plt.xlabel(u[i])
plt.show()
```

We were provided with train and test dataset. We split the train dataset into one into train and validation subsets. Then the dataset was reshaped.

Convolutional Neural Network (CNN)
Convolutional Neural Network (CNN) is a class of deep neural network which is used for analyzing visual imagery. We have used the following CNN model.

```
[12] #model
     cnn_model = Sequential()
     cnn_model.add(Conv2D(filters=32,kernel_size=3, activation='relu', input_shape=im_shape))
     cnn_model.add(MaxPool2D((2, 2)))
     cnn_model.add(Dropout(0.15))
     cnn_model.add(Conv2D(filters=64, kernel_size=3, activation='relu'))
     cnn_model.add(MaxPool2D((2, 2)))
     cnn_model.add(Dropout(0.15))
     cnn_model.add(Flatten())
     cnn_model.add(Dense(units=128, activation='relu'))
     cnn_model.add(Dense(5, activation='softmax'))
     monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5, verbose=1, mode='auto',
           restore_best_weights=True)
```

```
cnn_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 5, 5, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1600)              0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               204928
_________________________________________________________________
dense_2 (Dense)              (None, 5)                 645
=================================================================
Total params: 224,389
Trainable params: 224,389
Non-trainable params: 0
_________________________________________________________________
```

Convolutional Layer

Computers images are expressed as a matrix of pixels (NxNxd) — (height by width by depth). Images can use three channels (RGB) or one channel (Grayscale). In our dataset, the images are in grayscale.

CNN Filters:
CNN uses a set of learnable filters. The filters detect the presence of a specific features in the original image. It is usually expressed as a matrix (MxMx1) for grayscale image. It has the smaller dimension but the same depth compared to input file. The filter is convolved across the input image (Both across the width and height) to give the activation map. The Conv2D creates a convolutional layer. For example, in the first convolution layer we create 32 filters of size 3x3. The number of filters increase as we go deeper into the network.

Activation Function
Activation function is placed at the end or in between a neural network. It decides whether a neuron will fire or not. There are various types of activation function. For example:

- Sigmoid
- Relu
- Softmax

But, I have used mostly Relu and softmax.

Reasons for using Relu
Relu does not activate all neurons at the same time. It converts all negative inputs to zero deactivating the neurons. Since few neurons are activated per time, it is computationally efficient. It also converges six times faster than tanh and sigmoid activation functions.

Reasons for using Softmax just before output
Softmax is implemented through a neural network layer just before the output layer. The same number of nodes in the Softmax layer must be same as the output layer. Softmax extends the idea of logistic regression into a multi-class world. It assigns decimal probabilities to each class in a multi-class problem.

Pooling Layer
Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network. Pooling layer operates on each feature map independently. Pooling layer controls overfitting by progressively reducing the spatial size of the

network. The main use of pooling is to downsample the feature map while keeping the important information. In my model, I have used Max-pooling.

Max-pooling
Max-pooling takes out the maximum from a pool. Filters are silded through the input image and at every stride, the maximum value is selected. MaxPool2D creates a maxpooling layer, the only argument is the window size. We use a 2x2 window as it's the most common. In convolutional layer, the depth changes but it remains unaltered in max-pooling.

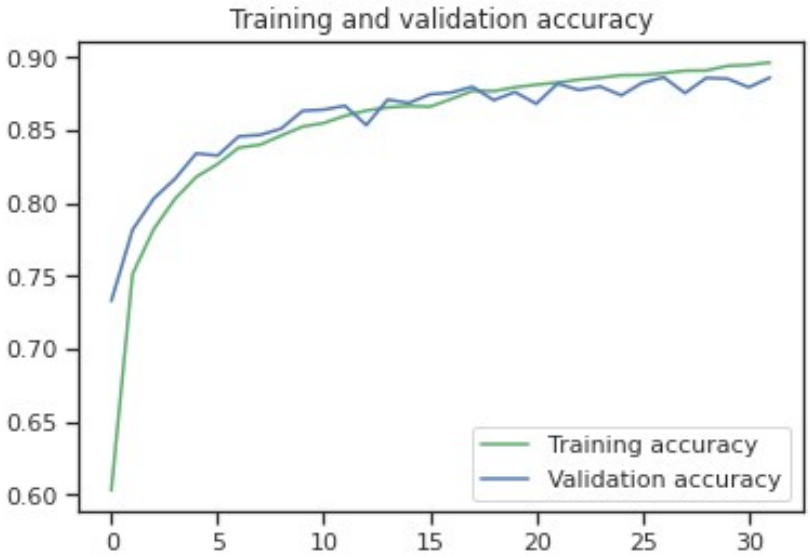
Flatten: In between the convolutional layer and the fully connected layer, there is a 'Flatten' layer. It transforms a two-dimensional matrix of features into a vector that can be fed into a fully connected neural network classifier. Flattening is simply arranging the 3D volume of numbers into a 1D vector.

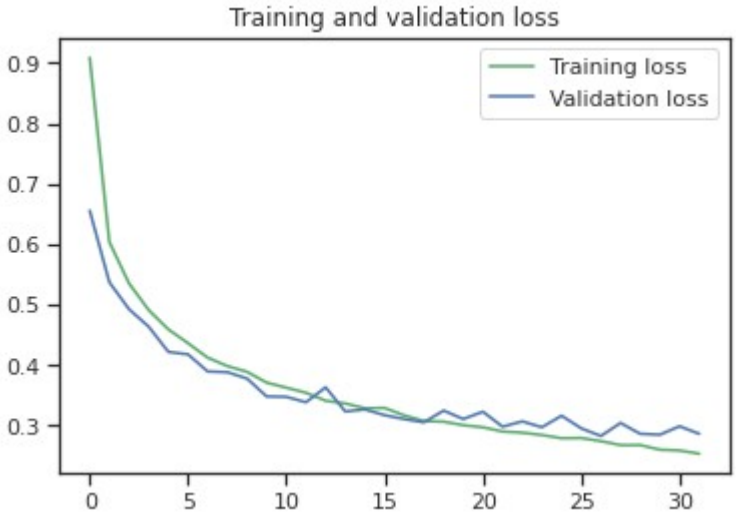
Dropout: Dropout is used for regularization in deep neural networks. Dropout is used to prevent overfitting. During training time, at each iteration, a neuron is temporarily "dropped" with probability p. We used a dropout rate of 0.15. It prevents the network to be too dependent on a small number of neurons and forces every neuron to be able to operate independently. Dropout can be applied on input or hidden nodes but not on the output nodes.

Optimizer: Here adam optimizer has been used. Adam is an adaptive learning rate optimization algorithm that's been designed specifically for training deep neural networks.

Accuracy: The validation accuracy is around 88.62%. The prediction for test dataset was submitted to kaggle and got an accuracy of 88.57%.

Plots: The loss and accuracy curves look as follows





There is no apparent overfitting since training loss and validation loss are roughly the same. Training accuracy and validation accuracy are also very close to each other. This shows that our model is doing well on unseen

data. To avoid overfitting, we used “EarlyStopping”. We implemented early stopping as a callback function.

```
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5, verbose=1, mode='auto',
      restore_best_weights=True)
```

In our code, we included EarlyStopping(monitor='val_loss', patience=5) to define that we wanted to monitor validation loss at each epoch and after the validation loss has not improved after five epochs, training is interrupted. We also store weight of the model having the best performance.

The confusion matrix of the model is given below:

```
array([[2265,   97,   14,    1,    2],
       [ 100, 2173,  122,   24,    5],
       [  12,  165, 2008,  139,   40],
       [   0,   37,  196, 1930,  204],
       [   4,    8,   39,   94, 2321]])
```

The classification report of the model is given below:

	precision	recall	f1-score	support
class 0	0.94	0.96	0.95	2379
class 1	0.88	0.89	0.88	2424
class 2	0.85	0.82	0.84	2364
class 3	0.83	0.86	0.85	2367
class 4	0.93	0.90	0.92	2466
accuracy			0.89	12000
macro avg	0.89	0.89	0.89	12000
weighted avg	0.89	0.89	0.89	12000

So 94% of the predictions for class 0 are actually of the predicted class, and 6% are actually of the other classes. Recall is the proportion of the true positives that are identified as such. This means that the model is correctly identifying 96% of the class 0s, but only 82% of the class 3s. Similarly other classes can be explained.
Class 0 has the highest precision and class 3 has the worst precision.

Runtime performance for training and testing:

- **Training performance:** It took 48 epochs to complete the training and each epoch took 2 seconds. So the total training time is around 96 seconds.
- **Testing performance:** For prediction it took around 0.8 seconds

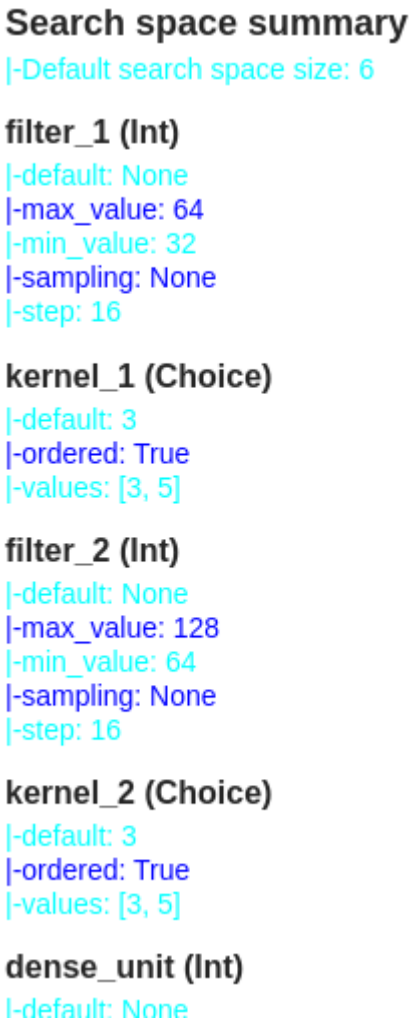
Fine-tuning CNN Model

In a keras tuner, we determined which hyperparameter combinations should be tested. A certain number of hyperparameter combinations were generated using iteration loops using a keras library function. Each model was checked for accuracy using a validation set. The model giving the highest accuracy was selected.

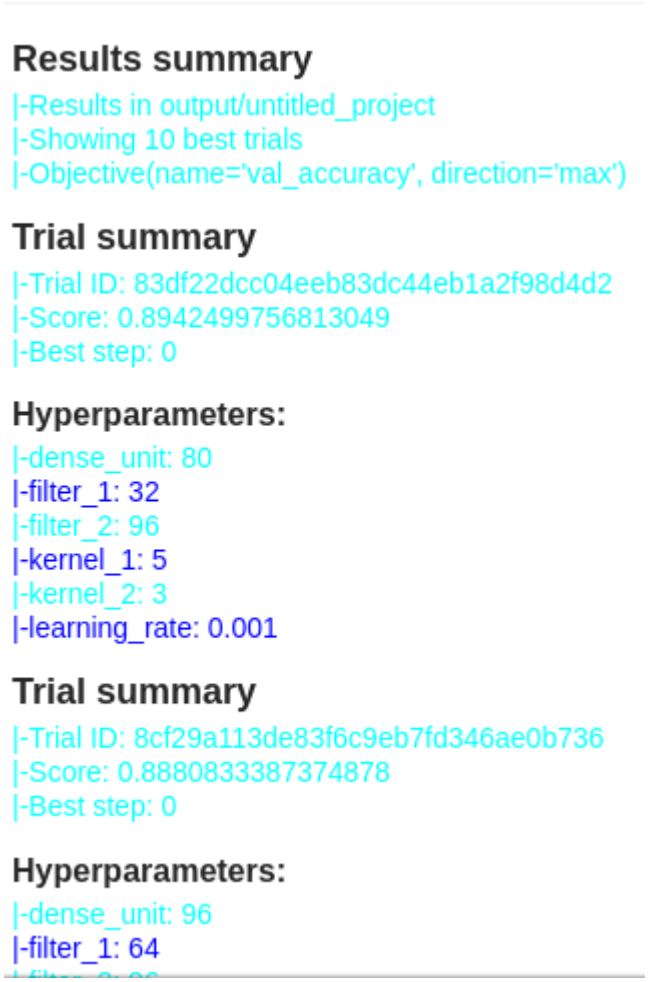
It takes an argument hp from which we can sample hyperparameters, such as hp.Int('filter_1', min_value=32, max_value=64, step=16) (an integer from a certain range).

```
#Defining model for keras tuner
from tensorflow import keras
from tensorflow.keras import layers
from kerastuner.tuners import RandomSearch
from tensorflow.keras.callbacks import EarlyStopping
def build_model(hp):
  cnn_model = keras.Sequential()
  cnn_model.add(keras.layers.Conv2D(filters=hp.Int('filter_1',min_value=32,max_value=64,step=16),
                                    kernel_size=hp.Choice('kernel_1', values = [3,5]),
                                    activation='relu', input_shape=im_shape))
  cnn_model.add(keras.layers.MaxPool2D((2, 2)))
  cnn_model.add(keras.layers.Dropout(0.15))
  cnn_model.add(keras.layers.Conv2D(filters=hp.Int('filter_2', min_value=64, max_value=128, step=16),
                                    kernel_size=hp.Choice('kernel_2', values = [3,5]),
                                    activation='relu'))
  cnn_model.add(keras.layers.MaxPool2D((2, 2)))
  cnn_model.add(keras.layers.Dropout(0.15))
  cnn_model.add(keras.layers.Flatten())
  cnn_model.add(keras.layers.Dense(units=hp.Int('dense_unit', min_value=32, max_value=128, step=16),
                                   activation='relu'))
  cnn_model.add(keras.layers.Dense(5, activation='softmax'))
  cnn_model.compile(optimizer=keras.optimizers.Adam(hp.Choice('learning_rate',
                      values=[1e-2, 1e-3, 1e-4])),
                    loss='sparse_categorical_crossentropy',
                    metrics=['accuracy'])
  return cnn_model
```

Some of the portion of search space summary is given below:



We also used “EarlyStopping” here. Some of the results from result summary is given below:



Accuracy: The validation accuracy after using the tuner is around 89% which is larger than our previous model. The prediction for test dataset was submitted to kaggle and got an accuracy of 89.78%.

The confusion matrix of the model is given below:

```
array([[2305,   60,   13,    0,    1],
       [ 164, 2105,  115,   35,    5],
       [  20,  168, 1993,  147,   36],
       [   0,   33,  195, 1997,  142],
       [   5,    7,   30,  142, 2282]])
```

The classification report of the best model is shown below:

	precision	recall	f1-score	support
class 0	0.94	0.97	0.95	2379
class 1	0.89	0.88	0.89	2424
class 2	0.87	0.82	0.85	2364
class 3	0.85	0.86	0.85	2367
class 4	0.92	0.93	0.93	2466
accuracy			0.89	12000
macro avg	0.89	0.89	0.89	12000
weighted avg	0.89	0.89	0.89	12000

So 94% of the predictions for class 0 are actually of the predicted class, and 6% are actually of the other classes. Recall is the proportion of the true positives that are identified as such. This means that the model is correctly identifying 97% of the class 0s, but only 82% of the class 2s. Similarly other classes can be explained.

Class 0 has the highest precision and class 3 has the worst precision which is similar to our previous classification report.

Runtime performance for training and testing:

- **Training performance:** It took 37 epochs to complete the training and each epoch took 5 seconds. So the total training time is around 185 seconds.
- **Testing performance:** For prediction it took around 0.3 seconds

Data Augmentation

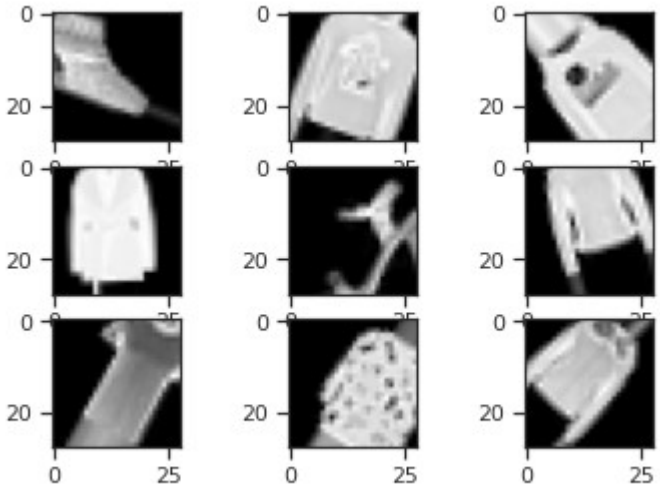
A large dataset is crucial for improving the performance of the deep learning model. However, we can improve the performance of the model by augmenting the data we already have.

Data augmentation techniques include affine transformations, perspective transformations, contrast changes, gaussian noise, dropout of regions, hue/saturation changes, cropping/padding, blurring etc. We have used the following transformations:

```
from keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(
    featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    zca_whitening=False,
    rotation_range=45,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True,
    vertical_flip=False)

datagen.fit(X_train)
```

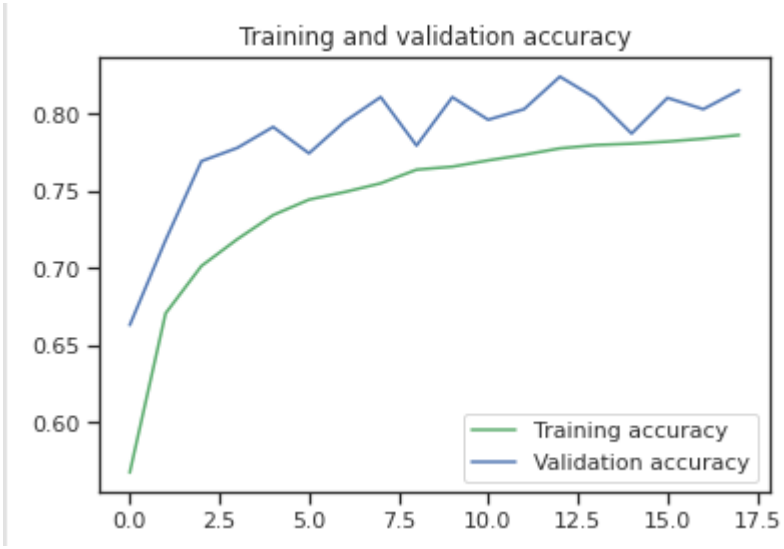
Some of the augmented images are also shown below:



In above images, various effects of data augmentation (rotation, width shift, height shift, horizontal flip etc.) is prevalent. We fed this data to a CNN model which is similar to previous CNN model.

In the real-world scenario, we may have a dataset of images taken in a limited set of conditions. But our target application may exist in a variety of conditions, such as different orientation, location, scale, brightness etc. We account for these situations by training our neural network with additional synthetically modified data.

Plots: The loss and accuracy curves look as follows



There is no apparent overfitting since training loss and validation loss are roughly the same. Training accuracy and validation accuracy are also very close to each other. This shows that our model is doing well on unseen data. To avoid overfitting, we used “EarlyStopping”. We implemented early stopping as a callback function.

Accuracy: The validation accuracy after using the data augmentation is around 81.52% which is smaller than our previous model. The prediction

for test dataset was submitted to kaggle and got an accuracy of 81.96%. Sometimes shifting and scaling forces overgeneralization. Overgeneralization decreases the performance because the network wastes its predictive capacity learning about irrelevant scenarios. It also depends on camera angle of the test dataset. If the angle at which the test photos were taken remains the same as the train dataset, the flipping and rotating the image does not help to increase the accuracy.

The confusion matrix of the model is given below:

```
array([[2146,  147,   85,    0,    1],
       [ 153, 1949,  226,   82,   14],
       [  31,  228, 1731,  324,   50],
       [   0,   57,  250, 1832,  228],
       [   3,   11,   77,  242, 2133]])
```

The classification report of the best model is shown below:

	precision	recall	f1-score	support
class 0	0.92	0.90	0.91	2379
class 1	0.81	0.80	0.81	2424
class 2	0.73	0.73	0.73	2364
class 3	0.74	0.77	0.76	2367
class 4	0.88	0.86	0.87	2466
accuracy			0.82	12000
macro avg	0.82	0.82	0.82	12000
weighted avg	0.82	0.82	0.82	12000

So 92% of the predictions for class 0 are actually of the predicted class, and 8% are actually of the other classes. Recall is the proportion of the true positives that are identified as such. This means that the model is correctly identifying 90% of the class 0s, but only 77% of the class 3s. Similarly other classes can be explained.
Class 0 has the highest precision and class 2 has the worst precision which is similar to our previous classification report.

Runtime performance for training and testing:

- **Training performance:** It took 18 epochs to complete the autoencoder training and each epoch took 13 seconds. So the total training time is around 234 seconds.
- **Testing performance:** For prediction it took around 0.32 seconds

Autoencoders

The main aim of autoencoders is to copy the input to output. This is accomplished by compressing the input into a latent space and then reconstructing from this. Every autoencoders have two parts:

Encoder: In this part of the network, the input is compressed into a latent space. It can be expressed by h=f(x)

Decoder: Here reconstruction from the latent space occurs. It is expressed by r=g(h)

Here, we want to keep h and r as close as possible. Autoencoders can learn data projections that are more interesting than PCA or other basic techniques.

Here we train a convolutional autoencoder and use the encoder part of the autoencoder combined with fully connected layers to recognize a new unknown sample.

```
def encoder(input_image):
    conv_lyr_1 = Conv2D(32, (3, 3), activation='relu', padding='same')(input_image)
    conv_lyr_1 = BatchNormalization()(conv_lyr_1)
    conv_lyr_1 = Conv2D(32, (3, 3), activation='relu', padding='same')(conv_lyr_1)
    conv_lyr_1 = BatchNormalization()(conv_lyr_1)
    pool_lyr_1 = MaxPooling2D(pool_size=(2, 2))(conv_lyr_1)
    conv_lyr_2 = Conv2D(64, (3, 3), activation='relu', padding='same')(pool_lyr_1)
    conv_lyr_2 = BatchNormalization()(conv_lyr_2)
    conv_lyr_2 = Conv2D(64, (3, 3), activation='relu', padding='same')(conv_lyr_2)
    conv_lyr_2 = BatchNormalization()(conv_lyr_2)
    pool_lyr_1  = MaxPooling2D(pool_size=(2, 2))(conv_lyr_2)
    conv_lyr_3  = Conv2D(128, (3, 3), activation='relu', padding='same')(pool_lyr_1 )
    conv_lyr_3  = BatchNormalization()(conv_lyr_3 )
    conv_lyr_3  = Conv2D(128, (3, 3), activation='relu', padding='same')(conv_lyr_3 )
    conv_lyr_3  = BatchNormalization()(conv_lyr_3 )
    conv_lyr_4 = Conv2D(256, (3, 3), activation='relu', padding='same')(conv_lyr_3 )
    conv_lyr_4 = BatchNormalization()(conv_lyr_4)
    conv_lyr_4 = Conv2D(256, (3, 3), activation='relu', padding='same')(conv_lyr_4)
    conv_lyr_4 = BatchNormalization()(conv_lyr_4)
    return conv_lyr_4
```

```
def decoder(conv_lyr_4):
    conv_lyr_5 = Conv2D(128, (3, 3), activation='relu', padding='same')(conv_lyr_4)
    conv_lyr_5 = BatchNormalization()(conv_lyr_5)
    conv_lyr_5 = Conv2D(128, (3, 3), activation='relu', padding='same')(conv_lyr_5)
    conv_lyr_5 = BatchNormalization()(conv_lyr_5)
    conv_lyr_6 = Conv2D(64, (3, 3), activation='relu', padding='same')(conv_lyr_5)
    conv_lyr_6 = BatchNormalization()(conv_lyr_6)
    conv_lyr_6 = Conv2D(64, (3, 3), activation='relu', padding='same')(conv_lyr_6)
    conv_lyr_6 = BatchNormalization()(conv_lyr_6)
    up_scale_1 = UpSampling2D((2,2))(conv_lyr_6)
    conv_lyr_7 = Conv2D(32, (3, 3), activation='relu', padding='same')(up_scale_1)
    conv_lyr_7 = BatchNormalization()(conv_lyr_7)
    conv_lyr_7 = Conv2D(32, (3, 3), activation='relu', padding='same')(conv_lyr_7)
    conv_lyr_7 = BatchNormalization()(conv_lyr_7)
    up_scale_2 = UpSampling2D((2,2))(conv_lyr_7)
    decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(up_scale_2)
    return decoded
```

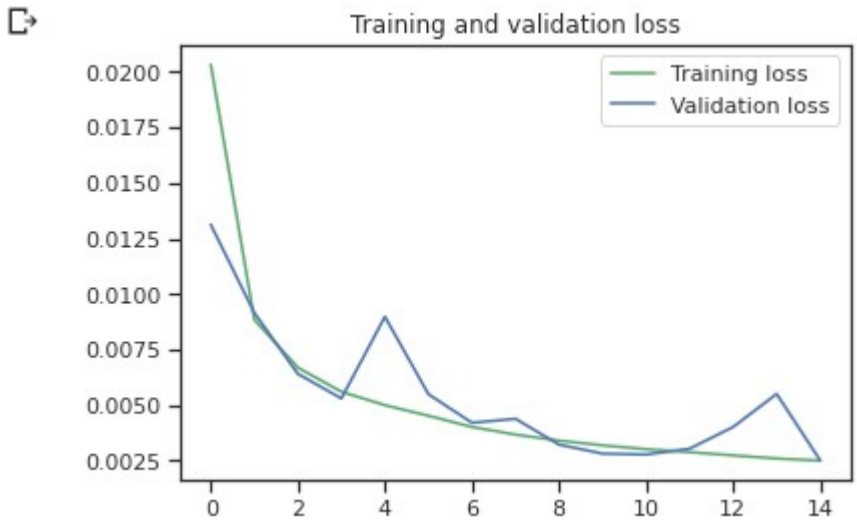
At first, we train convolutional autoencoder. For this, we send the data twice without sending the labels to the “train_test_split”.

```
X_train, X_valid, y_train, y_valid = train_test_split(X_train_sd.reshape(-1, 28,28, 1),
                                                      X_train_sd.reshape(-1, 28,28, 1),test_size=0.2,random_state=42)
```

Then we defined the encoder and decoder part. The number of filters, the filter size, the number of layers were decided based on intuition and taking help from the existing information.

```
[40] autoencoder = Model(input_image, decoder(encoder(input_image)))
     autoencoder.compile(loss='mean_squared_error', optimizer = RMSprop())
```

Here, we have used mean squared error to calculate the loss between the predicted output and ground truth pixel by pixel. RMSprop is an optimizer that utilizes the magnitude of recent gradients to normalize the gradients



Above figure (for the model without prediction capability) shows that the model is not overfitting since the validation loss and training loss both are in sync.

The labels were converted into one-hot encoding vectors so that we can feed to the algorithm.

Now we send the data with labels to the “train_test_split”.

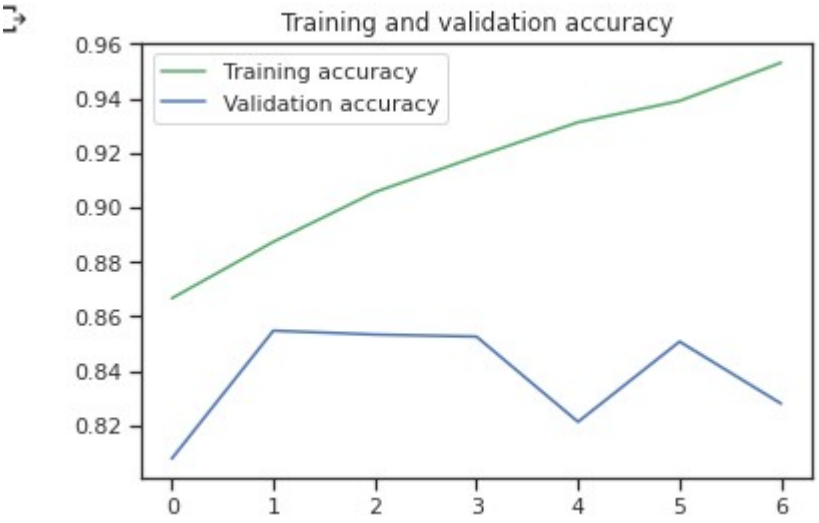
```
[ ] X_train_n,X_valid_n,y_train_n,y_valid_n = train_test_split(X_train_sd.reshape(-1, 28,28, 1),
                                                    train_Y_one_hot,test_size=0.2,random_state=13)
```

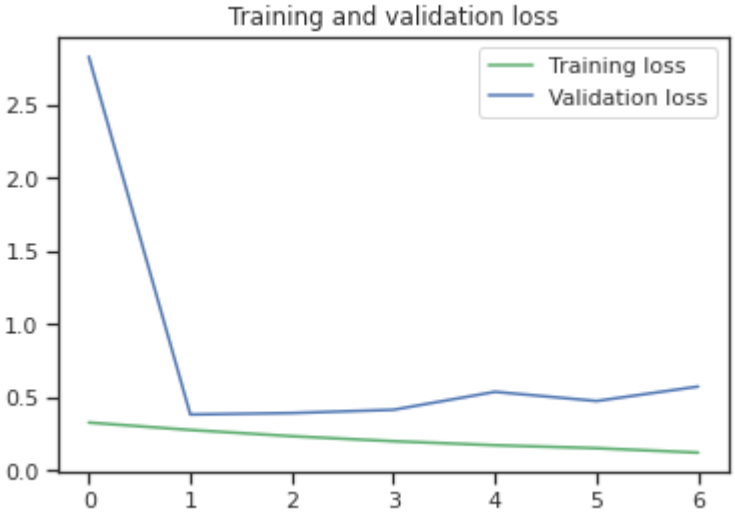
When we defined the model for classification, we used the same encoder. A fully connceted layer was defined that will be stacking up the encoder part.

```
[ ]  def fully_conn_lyr(input_enco):
         flatten_1= Flatten()(input_enco)
         dense_1 = Dense(128, activation='relu')(flatten_1)
         output_lyr = Dense(num_classes, activation='softmax')(dense_1)
         return output_lyr
```

The weights of the encoder part of the autoencoder were loaded to the encoder function of the classification model. The classification model was than trained for 25 epochs.

Plots: The loss and accuracy curves look as follows





From the above two plots, it can be seen that the model is overfitting to some extend since there is a gap between the training and validation loss. Besides, there is difference between training accuracy and validation accuracy.

Accuracy: The validation accuracy is around 85.48%. The prediction for test dataset was submitted to kaggle and got an accuracy of 85.52%.

The confusion matrix of the model is given below:

```
array([[2223,  153,   27,    0,    3],
       [ 108, 2053,  192,   42,   13],
       [  12,  225, 1938,  212,   36],
       [   1,   31,  292, 1904,  189],
       [   4,    7,   43,  152, 2140]])
```

The classification report of the best model is shown below:

	precision	recall	f1-score	support
class 0	0.95	0.92	0.94	2406
class 1	0.83	0.85	0.84	2408
class 2	0.78	0.80	0.79	2423
class 3	0.82	0.79	0.81	2417
class 4	0.90	0.91	0.91	2346
accuracy			0.85	12000
macro avg	0.86	0.86	0.86	12000
weighted avg	0.86	0.85	0.85	12000

So 95% of the predictions for class 0 are actually of the predicted class, and 5% are actually of the other classes. Recall is the proportion of the true positives that are identified as such. This means that the model is correctly identifying 92% of the class 0s, but only 79% of the class 3s. Similarly other classes can be explained.
Class 0 has the highest precision and class 2 has the worst precision which is similar to our previous classification report.

Runtime performance for training and testing:

- **Training performance:** It took 15 epochs to complete the autoencoder training and each epoch took 23 seconds. Similarly, It took 7 epochs to complete the autoencoder classifier for training and each epoch took 16 seconds. So the total training time is around 457 seconds.
- **Testing performance:** For prediction it took around 1.44 seconds

Transfer learning:

Transfer learning is a popular method in computer vision because it allows us to build accurate models in a timesaving way. It is the idea of utilizing knowledge acquired for one task to solve related ones. Traditional learning create isolated models based on specific dataset, tasks and training. Here, we can not tranfer knowledge from one model to another. In transfer learning we can transfer the features, weights from previously trained models to newer one.

VGG-16

VGG-16 is a convolutional neural network architecture which has 16 layers. It's layers consists of Convolutional layers, Max Pooling layers, Activation layers, Fully connected layers. It has

- 13 convolutional layers
- 5 Max Pooling layers
- 3 Dense layers

VGG-16 network is trained on ImageNet dataset which has over 14 million images and 1000 classes, and acheives 92.7% top-5 accuracy. Keras provides access to VGG-16 along with a number of top-performing pre-trained models that were developed for image recognition tasks.

For implementing VGG-16 , we have to do a number of tasks.

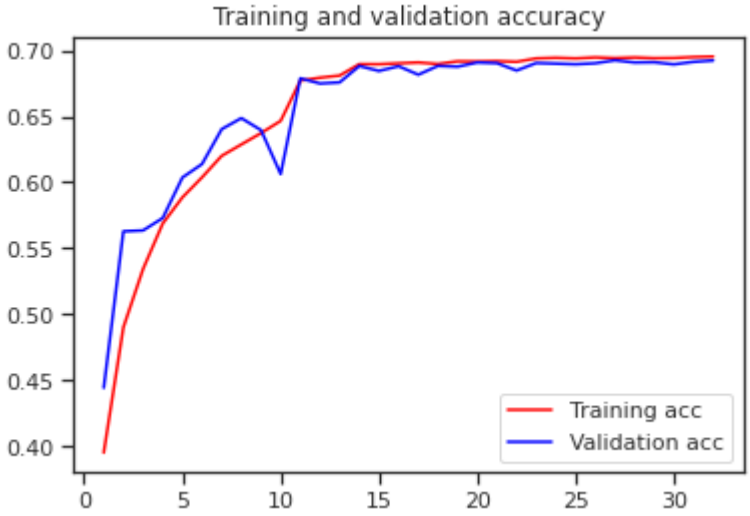
- Convert grayscale image to 3 channel images.
- We have to convert to our image to the size required by VGG-16 ie 48*48

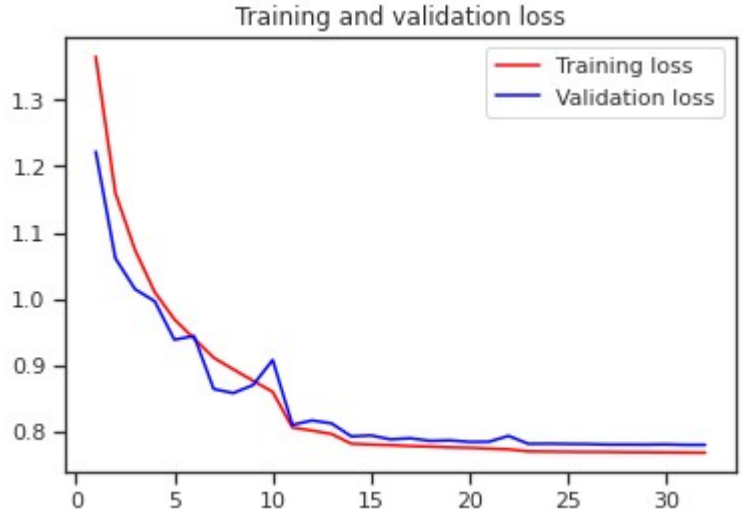
In our code we did the following things:

- Converted single channel dataset to three channel dataset as required by the VGG-16 model
- Reshaped the images to 48*48 as required by the VGG-16
- Normalized the dataset by dividing it by 255
- Used preprocess_input to adequate the image dataset to the format the model requires.
- We created a VGG-16 model and weights were defined from 'imagenet'
- We extraced the features so that it can be fed into the dense layer
- Incorporating reduced learning and early stopping for callback

```
[ ]  #base model of VGG16
     VGG_base = VGG16(weights='imagenet',
                      include_top=False,
                      input_shape=(y_dim, x_dim, chan)
                     )
     VGG_base.summary()
```

Plots: The loss and accuracy curves look as follows





There is no apparent overfitting since training loss and validation loss are roughly the same. Training accuracy and validation accuracy are also very close to each other. This shows that our model is doing well on unseen data. To avoid overfitting, we used “EarlyStopping”. We implemented early stopping as a callback function.

Accuracy: The validation accuracy is around 69.23% The prediction for test dataset was submitted to kaggle and got an accuracy of 70.16%.

The confusion matrix of the model is given below:

```
array([[2033,  320,   47,    1,    5],
       [ 324, 1606,  372,   43,   63],
       [ 128,  449, 1324,  398,  124],
       [  10,  127,  350, 1648,  282],
       [  55,   75,   90,  430, 1696]])
```

The classification report of the best model is shown below:

	precision	recall	f1-score	support
class 0	0.80	0.84	0.82	2406
class 1	0.62	0.67	0.64	2408
class 2	0.61	0.55	0.57	2423
class 3	0.65	0.68	0.67	2417
class 4	0.78	0.72	0.75	2346
accuracy			0.69	12000
macro avg	0.69	0.69	0.69	12000
weighted avg	0.69	0.69	0.69	12000

So 80% of the predictions for class 0 are actually of the predicted class, and 20% are actually of the other classes. Recall is the proportion of the true positives that are identified as such. This means that the model is correctly identifying 84% of the class 0s, but only 68% of the class 3s. Similarly other classes can be explained.
Class 0 has the highest precision and class 2 has the worst precision which is similar to our previous classification report.

Runtime performance for training and testing:

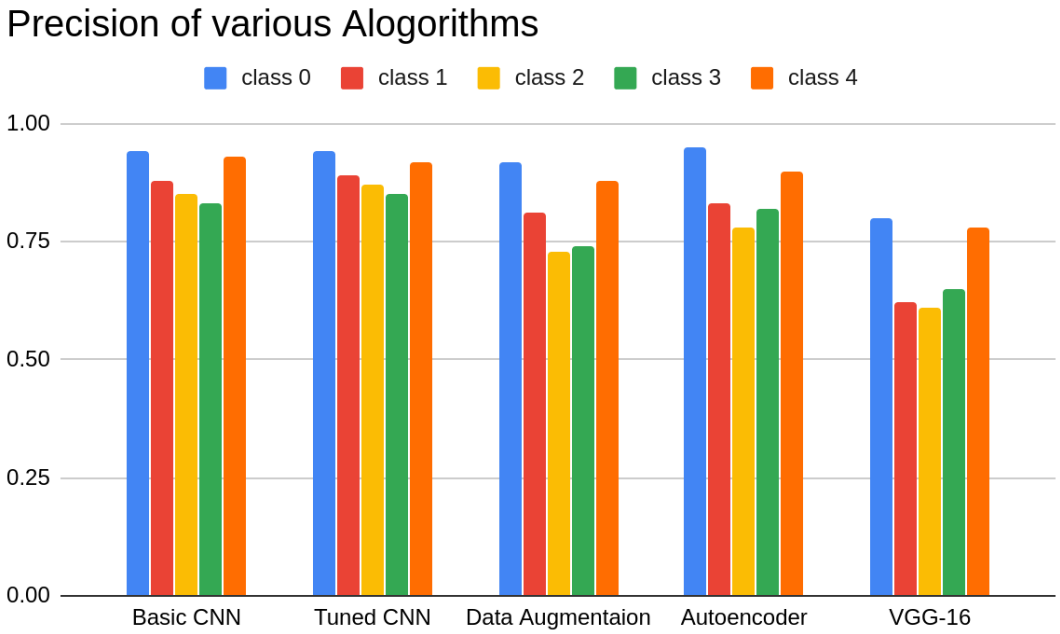
- **Training performance:** It took 32 epochs to complete the training and each epoch took 4 seconds. So the total training time is around 128 seconds.
- **Testing performance:** For prediction it took around 0.26 seconds

Kaggle Competition Score:
The highest accuracy in kaggle was 89.78% for our team. The highest score reported in the kaggle is around 92.4%. Here, the accuracy can depend on the following things:

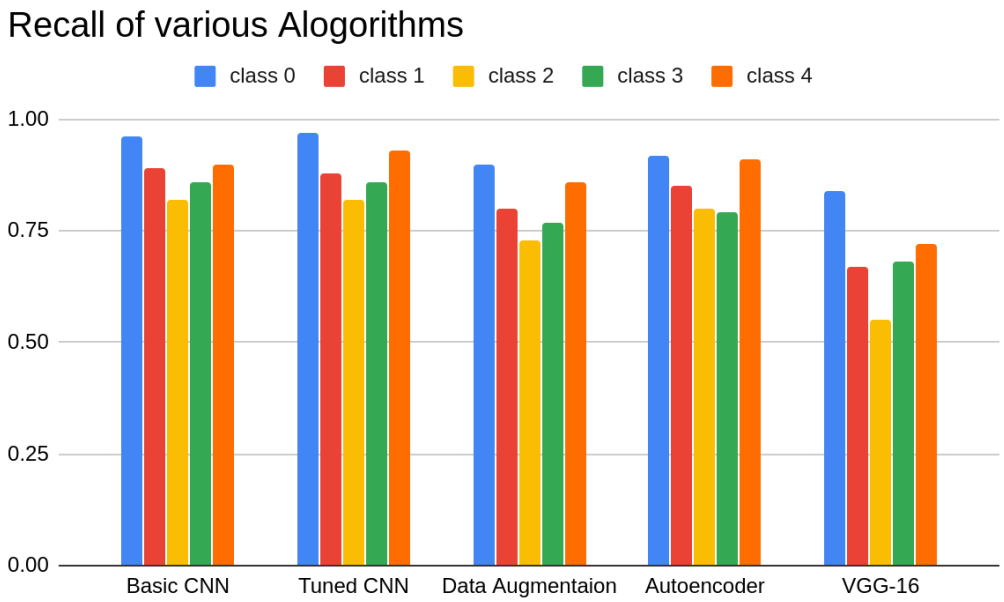
- Design variations

- train test split because more training data usually makes the prediction better for image. So variation in train test split can result in variation in accuracy.

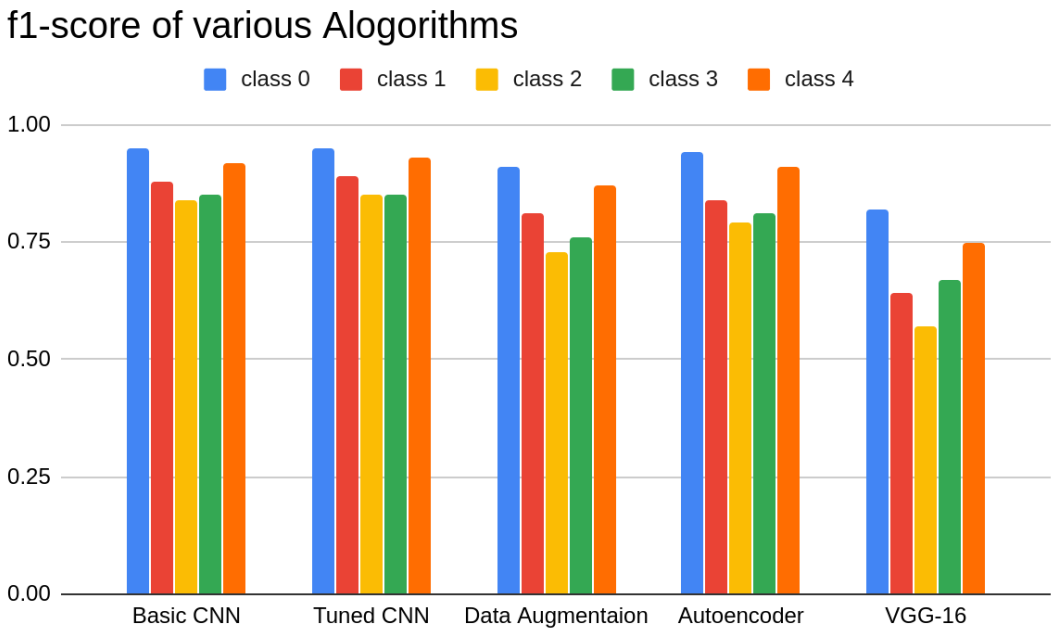
Analysis:

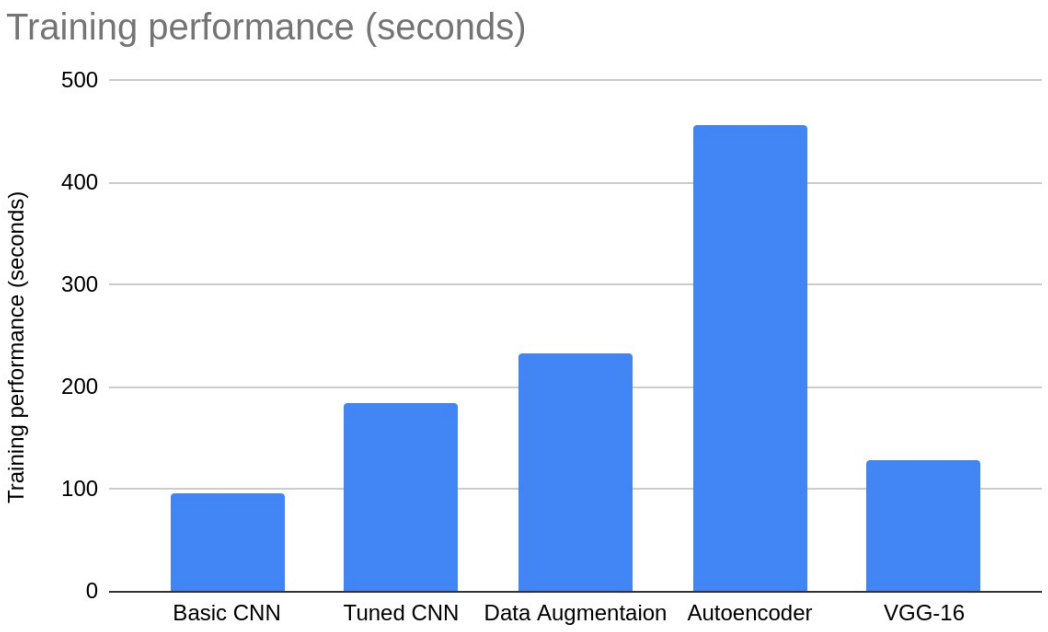


From the above graph we can see that Basic CNN and Tuned CNN is giving almost identical precision for every class. Data augmentation and Autoencoder model is giving good precision for both class 0 and class 4. Class 2 and class 3 is giving worst performance in all algorithms. VGG-16 gave worst precision for classes compared to other algorithms. This type of analysis is important if we want to know which algorithms is giving best precision for which class.

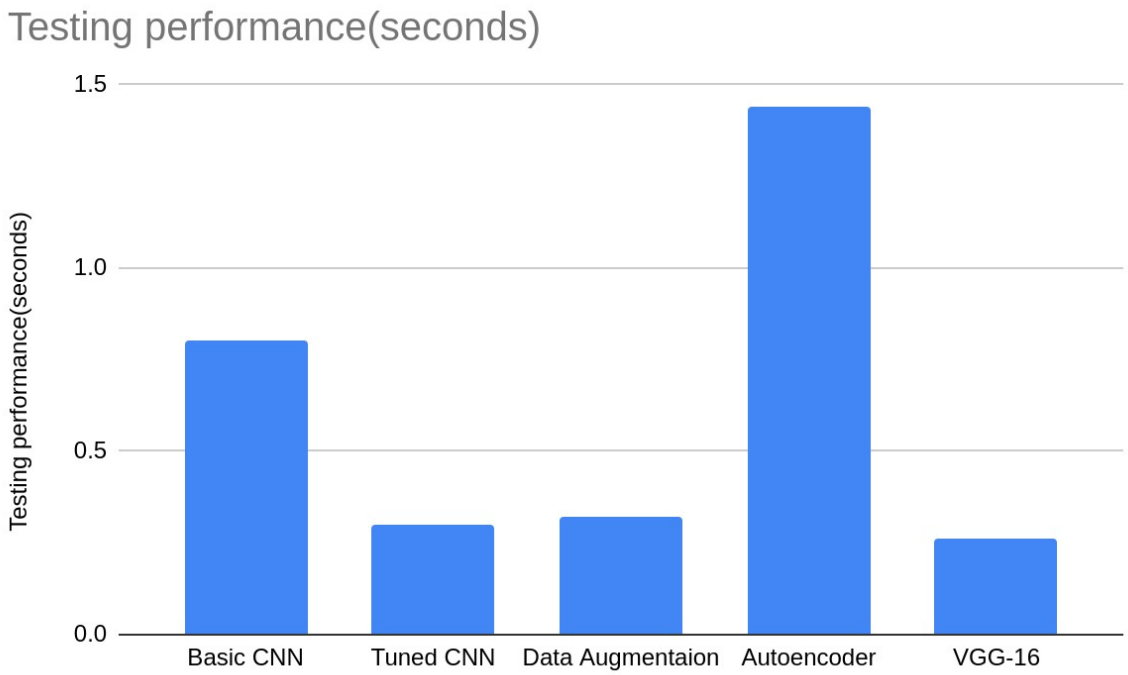


The case of recall is almost identical to the precision one. Basic CNN and Tuned CNN giving best recall while VGG-16 giving the worst for all classes.

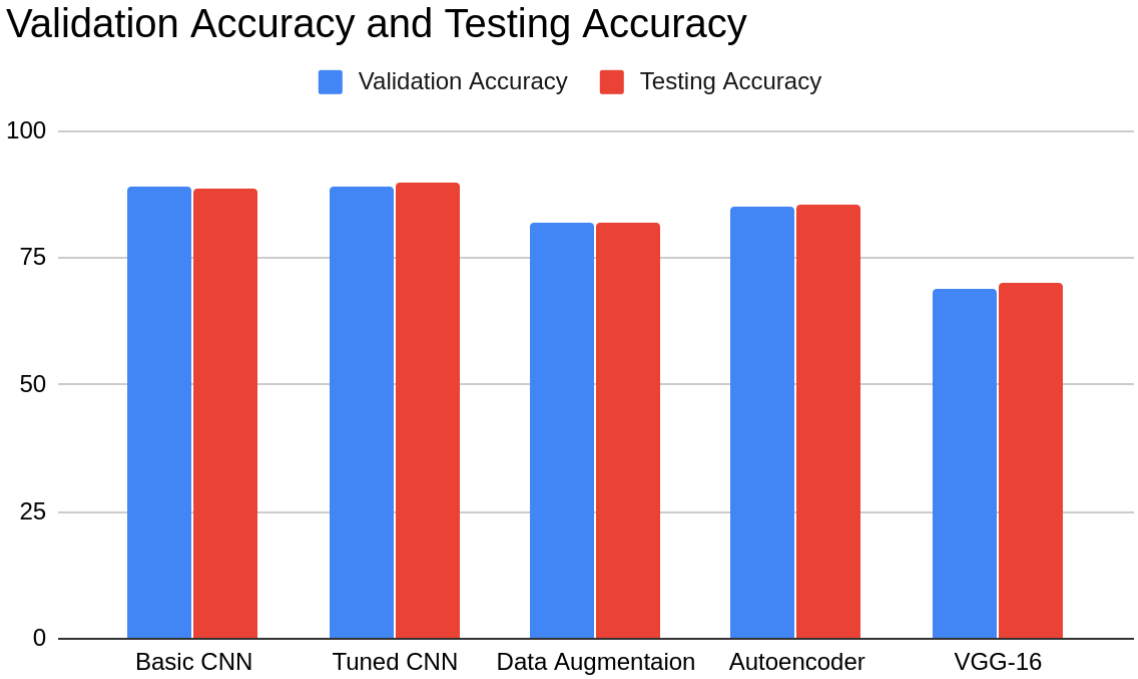




In terms of training time, autoencoder took the highest time since it had to train twice. First, an autoencoder with encoder and decoder were trained then a classifier model with only encoder was trained. Data augmentation model took the second place. It has higher train time because data augmentation creates new images so the training data also increases in size. VGG-16 has the second least training time because it is using transfer learning.



In case of testing (predicting), Autoencoder took the highest time. Basic CNN took almost half of the time compared to autoencoder. Other algorithms almost took the same time for prediction.



In terms of validation and testing (kaggle) accuracy, Basic CNN and Tuned CNN has almost identical accuracy. Tuned CNN has the highest accuracy compared to other algorithms. In case of all the algorithms, there is not much difference in terms of validation and testing (kaggle) accuracy. This shows that our algorithms are not overfitting which is essential for a good model.

Comparison of the different algorithms and parameters:

Algorithms	Important Parameters	Accuracy
Basic CNN	• 2 convolutional layer each followed by a Maxpool and dropout layer • 1 Flatten layer • 2 dense layer • Relu and softmax for activation • Adam optimizer • Early stopping was used to overcome overfitting.	88.62%
Tuned CNN	• Same as Basic CNN but here the hyper-parameters were optimized for higher accuracy	89%
Data Augmentaion	• Same as Tuned CNN • New training images were created using ImageDataGenerator • Rotation, width shift, height shift horizontal flip were used for new image creation	82%
Autoencoder	• Encoder and decoder were defined using convolutional layer, maxpooling layer, batch normalization, upsampling layer • Classification model was built using encoder followed by a dense layer • Early stopping was used to overcome overfitting.	85%
VGG-16	• Weights of the model was taken from imagenet • Adam optimizer was used • Early stopping was used to overcome overfitting.	69%