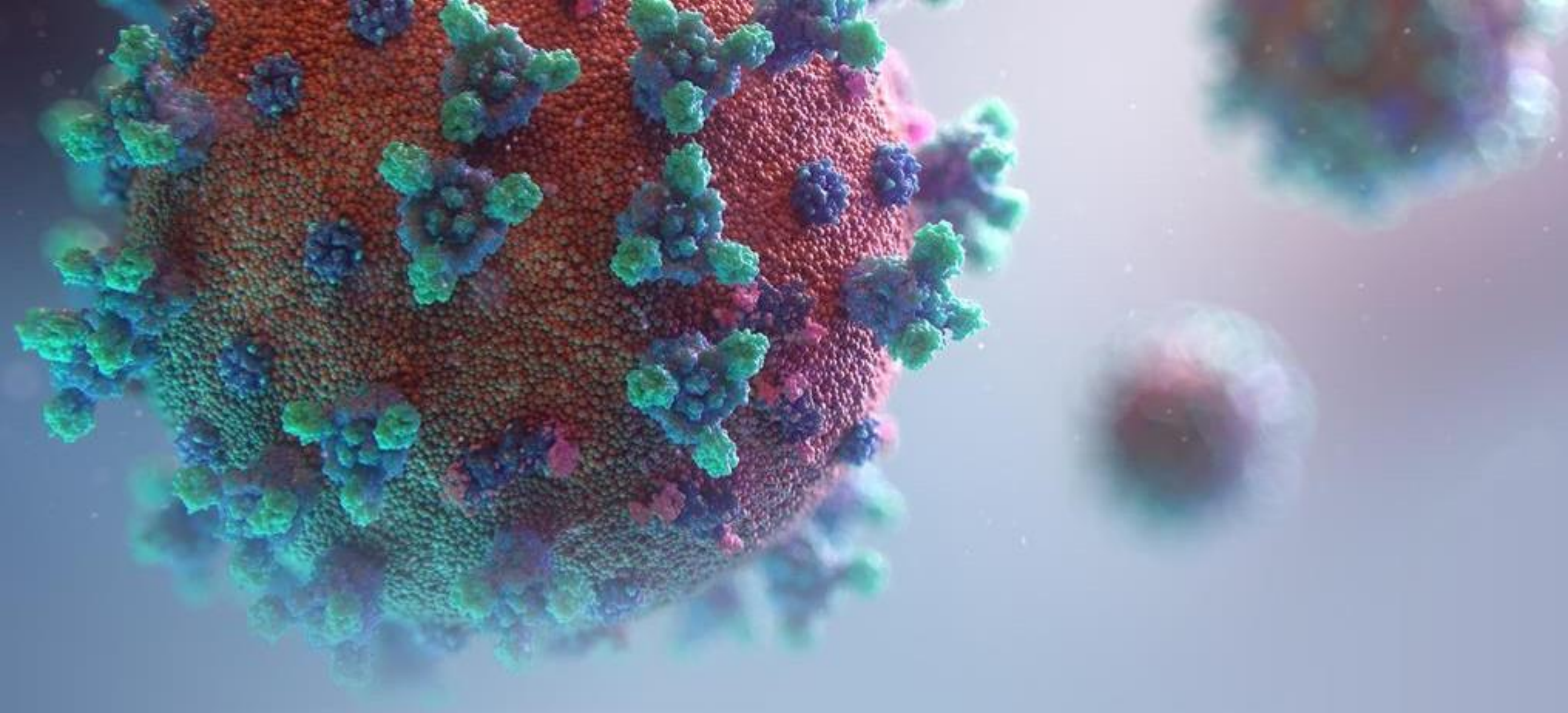


# Fashion MNIST

By: Yara Elzahy
ID: 20398570

# Problem Statement

**Zalando’s Fashion MNIST dataset** is comprised of 60,000 small square 28x28 pixel grayscale images of items of 10 types of clothing, such as shoes, dresses and trousers. The mapping of all 0-9 integers to class label is listed as follows:

- **0**: T-shirt/top
- **1**: Trouser
- **2**: Pullover
- **3**: Dress
- **4**: Coat
- **5**: Sandal
- **6**: Shirt
- **7**: Sneaker
- **8**: Bag
- **9**: Ankle boot

### Data Description:

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, which indicates how light or dark that pixel is, with larger numbers suggesting darker. This pixel value is an integer ranging from 0 to 255. There are 785 columns in the training and test data sets. The first column is made up of class labels as seen in the above figure and represents an article of clothing. The remaining columns contain the corresponding image's pixel values.

- Each row represents a distinct image.
- Column 1 contains the class label, whereas the remaining columns include pixel numbers (784 total).
- The pixel's darkness is represented by each value (1 to 255)

# Problem Formulation

- **Input**: 28 x 28 pixel grayscale images of items of 10 types of clothing, such as shoes and dresses.
- **Output**: predict the type of clothing (e.g. Bag, t-shirt, …etc.)
- **Deep learning Function:** Manipulating, analyzing, preprocessing and training the data.

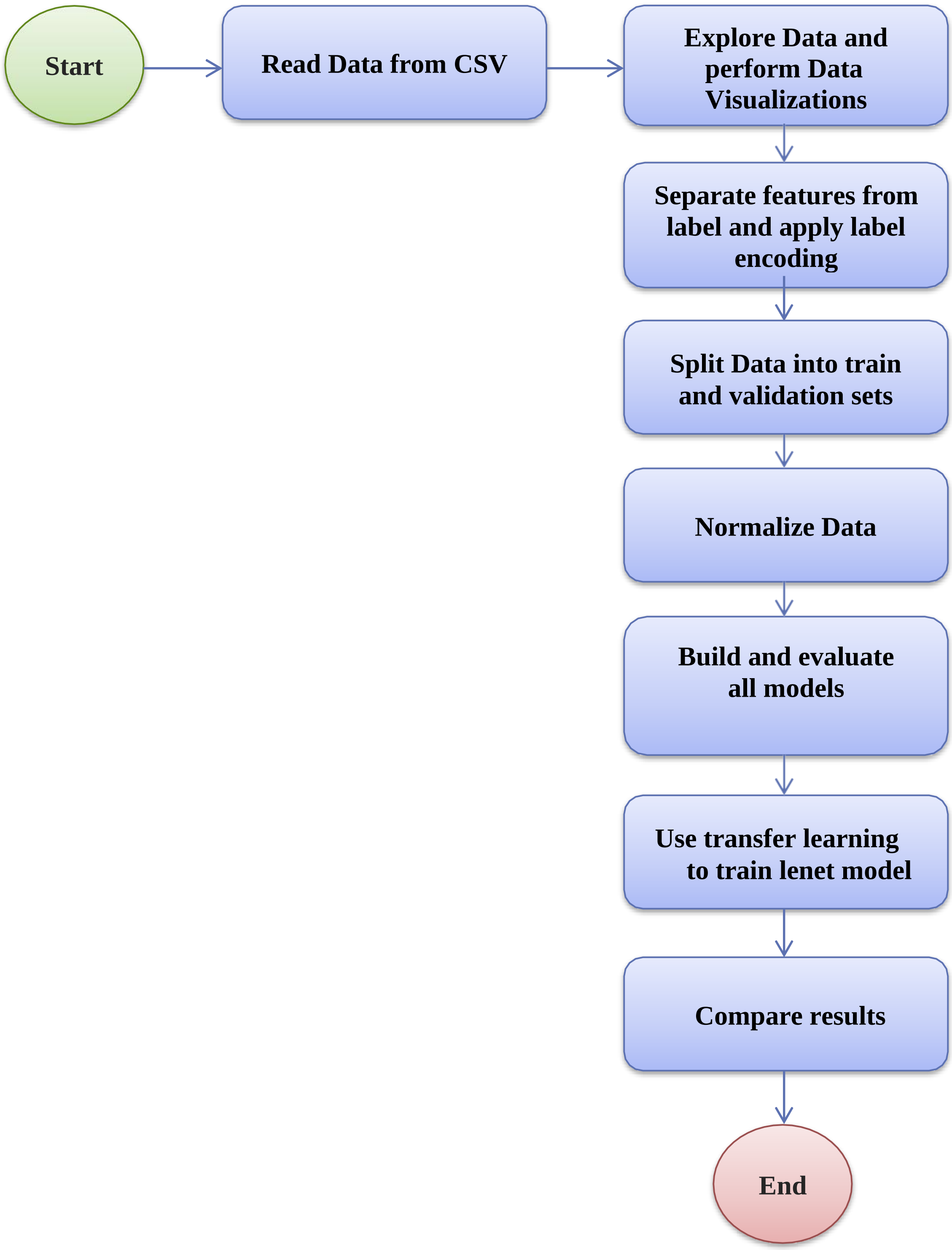
## Challenges:

1. Working with images
2. Adjust shape of training data to suit the lenet-5 architecture.
3. Choose the best hyper-parameters for the network.
4. Apply transfer learning.

## Impact:

Predicting the type of clothing (e.g. Bag, shirt, …etc.)

# Project Pipeline



# Part 1 – Data Preparation

## 1) Read and Display Training Data

```
[6] train = pd.read_csv('/content/fashion-mnist_train.csv')
    test = pd.read_csv('/content/fashion-mnist_test.csv')
```

```
[7] train.head()
```

| | label | pixel1 | pixel2 | pixel3 | pixel4 | ... | pixel780 | pixel781 | pixel782 | pixel783 | pixel784 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 |
| 1 | 9 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 |
| 2 | 6 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | ... | 1 | 0 | 0 | 0 | 0 |
| 4 | 3 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 |

5 rows × 785 columns

```
[8] train.shape, test.shape
```

```
((60000, 785), (10000, 785))
```

## 2) Data Preprocessing

### Clean the data

```
[11] train.isnull().sum().sort_values(ascending=False).sum()
```

```
0
```

There are no null values

## Check for duplicates

```
[12] # print the same of duplicates
     train.duplicated().sum()

     43
```

```
[13] # drop duplicates
     train.drop_duplicates(subset=None, keep="first", inplace=True)
```

```
[14] # print the same of duplicates
     train.duplicated().sum()

     0
```

Data contains neither duplicates nor null values.

**3) Data Visualization**

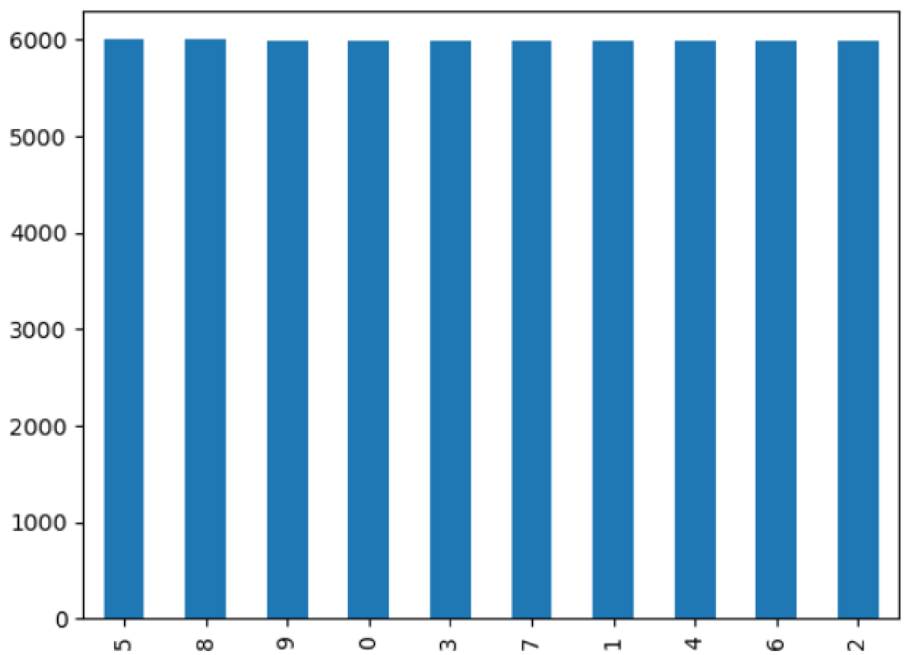



Fig.1.indicates balanced data

```python
# displaying heatmap
plt.show()
```

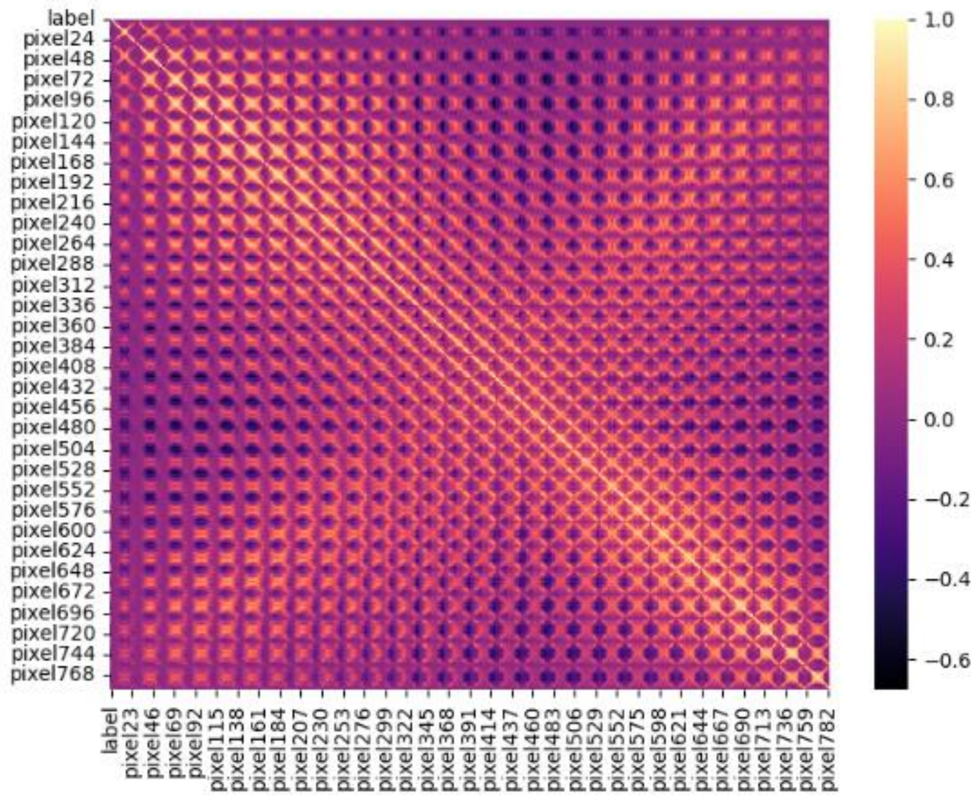
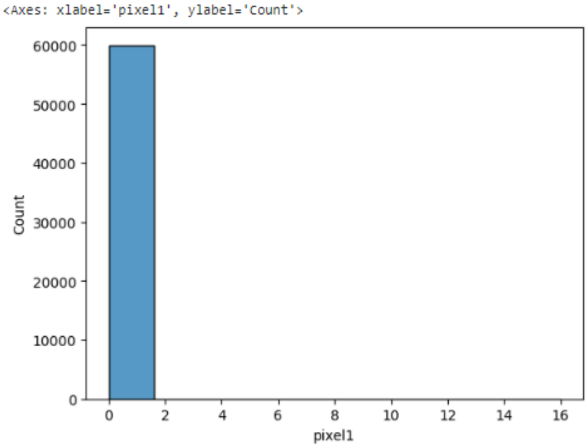



Fig.2. correlation matrix

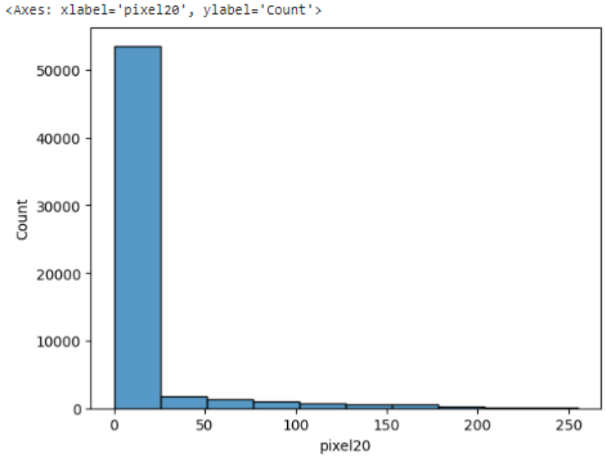
```python
sns.histplot(train['pixel1'], bins=10, kde=False)
```

<Axes: xlabel='pixel1', ylabel='Count'>




```python
sns.histplot(train['pixel20'], bins=10, kde=False)
```

<Axes: xlabel='pixel20', ylabel='Count'>



```python
# Image visualization
# assign the labels to its corresponding values from the data set
class_names = ['T_shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
plt.figure(figsize=(10,10))

for i in range(100):
    plt.subplot(10,10,i+1)
    plt.imshow(x_train[i].reshape(28,28))
    plt.title(class_names[int(y_train[i])])
    plt.axis('off')
plt.subplots_adjust(hspace=.7,wspace=0.8)
plt.show()
```


## 3) Encode labels using one hot encoding

```python
[28] from sklearn.preprocessing import OneHotEncoder

     one_hot_encoder = OneHotEncoder(sparse=False)
     train_labels_one_hot = one_hot_encoder.fit_transform(y_train.reshape(-1, 1))
     val_labels_one_hot = one_hot_encoder.transform(y_val.reshape(-1, 1))
```

```
/usr/local/lib/python3.9/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `s
  warnings.warn(
```

```python
[29] train_labels_one_hot
```

```
array([[0., 0., 0., ..., 1., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 1., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

```python
[30] val_labels_one_hot
```

```
array([[0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 1., 0., ..., 0., 0., 0.],
```

# Part 2 – Training a CNN neural network

Adjust the shapes of train and validation data so to use as an input to the leNet-5 model

```
[34] # reshape the train and validation data to be # of rows x 28 x 28
     x_train1 = x_train.reshape(-1,28,28)
     x_val1 = x_val.reshape(-1,28,28)
     x_train1.shape, x_val1.shape

     ((47965, 28, 28), (11992, 28, 28))
```

```
[35] # perform padding and normalization on the train and validation datasets so their shape is # of rows x 32 x 32
     x_train1 = tf.pad(x_train1, [[0, 0], [2,2], [2,2]])/255
     x_val1 = tf.pad(x_val1, [[0, 0], [2,2], [2,2]])/255
     x_train1.shape, x_val1.shape

     (TensorShape([47965, 32, 32]), TensorShape([11992, 32, 32]))
```

```
     # expand dimension so the train and validation datasets have an additional dimension
     x_train1 = tf.expand_dims(x_train1, axis=3, name=None)
     x_val1 = tf.expand_dims(x_val1, axis=3, name=None)
     x_train1.shape

     TensorShape([47965, 32, 32, 1])
```

```
[37] x_train1.shape[1:]

     TensorShape([32, 32, 1])
```

### Structure of the LeNet-5 Model

LeNet5 is a small network, it contains the basic modules of deep learning: convolutional layer, pooling layer, and full link layer. It is the basis of other deep learning models. Here we analyze LeNet5 in depth. At the same time, through example analysis, deepen the understanding of the convolutional layer and pooling layer.LeNet-5 Total seven layer, does not comprise an input, each containing a trainable parameters; each layer has a plurality of the Map the Feature, a characteristic of each of the input FeatureMap extracted by means of a convolution filter, and then each FeatureMap There are multiple neurons.

| Layer | | Feature Map | Size | Kernel Size | Stride | Activation |
|---|---|---|---|---|---|---|
| Input | Image | 1 | 32x32 | - | - | - |
| 1 | Convolution | 6 | 28x28 | 5x5 | 1 | tanh |
| 2 | Average Pooling | 6 | 14x14 | 2x2 | 2 | tanh |
| 3 | Convolution | 16 | 10x10 | 5x5 | 1 | tanh |
| 4 | Average Pooling | 16 | 5x5 | 2x2 | 2 | tanh |
| 5 | Convolution | 120 | 1x1 | 5x5 | 1 | tanh |
| 6 | FC | - | 84 | - | - | tanh |
| Output | FC | - | 10 | - | - | softmax |

### Input Layer

The first is the data INPUT layer. The size of the input image is uniformly normalized to 32 * 32.

Note: This layer does not count as the network structure of LeNet-5. Traditionally, the input layer is not considered as one of the network hierarchy.

### C1 layer-convolutional layer:

- **Input picture**: 32 * 32
- **Convolution kernel size**: 5 * 5
- **Convolution kernel types**: 6
- **Output feature-map size**: 28 * 28 (32-5 + 1) = 28
- **Number of neurons**: 28 *28* 6
- **Trainable parameters**: (5 *5 + 1)* 6 (5 * 5 = 25 unit parameters and one bias parameter per filter, a total of 6 filters)
- **Number of connections**: (5 *5 + 1)* 6 *28* 28 = 122304

### S2 layer-pooling layer (down-sampling layer):

- **Input**: 28 * 28
- **Sampling area**: 2 * 2
- **Sampling method**: 4 inputs are added, multiplied by a trainable parameter, plus a trainable offset. Results via sigmoid
- **Sampling type**: 6
- **Output featureMap size**: 14 * 14 (28/2)
- **Number of neurons**: 14 *14* 6
- **Trainable parameters**: 2 * 6 (the weight of the sum + the offset)
- **Number of connections**: (2 *2 + 1)* 6 *14* 14
- The size of each feature map in S2 is 1/4 of the size of the feature map in C1.

### C3 layer-convolutional layer:

- **Input**: all 6 or several feature map combinations in S2
- **Convolution kernel size**: 5 * 5
- **Convolution kernel type**: 16
- **Output featureMap size**: 10 * 10 (14-5 + 1) = 10
- **The trainable parameters are:** 6 *(3* 5 *5 + 1) + 6* (4 *5* 5 + 1) + 3 *(4* 5 *5 + 1) + 1* (6 *5* 5 +1) = 1516
- **Number of connections**: 10 *10* 1516 = 151600

### S4 layer-pooling layer (downsampling layer)

- **Input**: 10 * 10
- **Sampling area**: 2 * 2
- **Sampling method**: 4 inputs are added, multiplied by a trainable parameter, plus a trainable offset. Results via sigmoid
- **Sampling type**: 16
- **Output featureMap size**: 5 * 5 (10/2)
- **Number of neurons**: 5 *5* 16 = 400
- **Trainable parameters**: 2 * 16 = 32 (the weight of the sum + the offset)
- **Number of connections**: 16 *(2* 2 + 1) *5* 5 = 2000
- The size of each feature map in S4 is 1/4 of the size of the feature map in C3

### C5 layer-convolution layer

- **Input**: All 16 unit feature maps of the S4 layer (all connected to s4)
- **Convolution kernel size**: 5 * 5
- **Convolution kernel type**: 120
- **Output feature-Map size**: 1 * 1 (5-5 + 1)
- **Trainable parameters / connection**: 120 *(16* 5 * 5 + 1) = 48120

### F6 layer-fully connected layer

- **Input**: c5 120-dimensional vector
- **Calculation method**: calculate the dot product between the input vector and the weight vector, plus an offset, and the result is output through the sigmoid function.
- **Trainable parameters**: 84 * (120 + 1) = 10164

### Output layer-fully connected layer

The output layer is also a fully connected layer, with a total of 10 nodes, which respectively represent the numbers 0 to 9, and if the value of node i is 0, the result of network recognition is the number i. A radial basis function (RBF) network connection is used. Assuming x is the input of the previous layer and y is the output of the RBF, the calculation of the RBF output is:

$$y_i = \sum_j \left(x_j - w_{ij}\right)^2$$

## Build the LeNet-5 Model

```python
# adjust the input shape to be 32 x 32 x 1 so it's suitable for the lenet-5 architecture
input_shape = x_train1.shape[1:]

# create_model function takes the input shape as an input parameter and builds the lenet-5 model
def create_model(input_shape):
    lenet_5_model = Sequential()
    lenet_5_model.add(Conv2D(filters=32, kernel_size=(5,5), padding='same', activation='relu', input_shape=input_shape))
    lenet_5_model.add(MaxPooling2D(strides=2))
    lenet_5_model.add(Conv2D(filters=48, kernel_size=(5,5), padding='valid', activation='relu'))
    lenet_5_model.add(MaxPooling2D(strides=2))
    lenet_5_model.add(Flatten())
    lenet_5_model.add(Dense(256, activation='relu'))
    lenet_5_model.add(Dense(84, activation='relu'))
    lenet_5_model.add(Dense(10, activation='softmax'))
    lenet_5_model.build()
    lenet_5_model.compile(optimizer=Adam(), loss=losses.categorical_crossentropy, metrics=['accuracy'])

    return lenet_5_model
```

```python
model = create_model(x_train1.shape[1:])
model.summary()
```

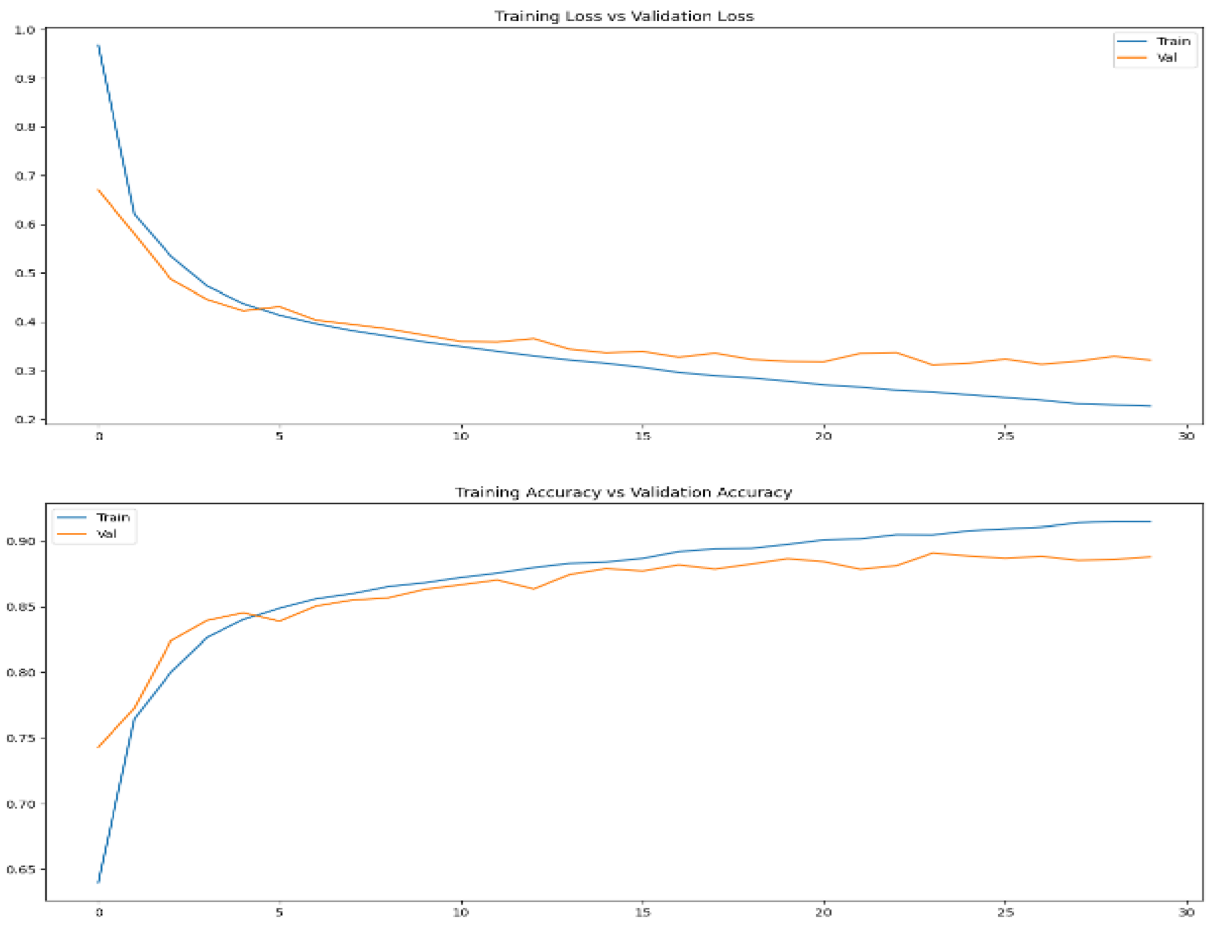
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 32)        832

 max_pooling2d (MaxPooling2D  (None, 16, 16, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 12, 12, 48)        38448

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 48)         0
 2D)

 flatten (Flatten)           (None, 1728)              0

 dense (Dense)               (None, 256)               442624

 dense_1 (Dense)             (None, 84)                21588

 dense_2 (Dense)             (None, 10)                850

=================================================================
Total params: 504,342
Trainable params: 504,342
Non-trainable params: 0
_________________________________________________________________
```

```python
history = model.fit(x_train1, train_labels_one_hot, batch_size=64, epochs=30, validation_data=(x_val1, val_labels_one_hot))
```

```
750/750 [==============================] - 16s 8ms/step - loss: 0.9618 - accuracy: 0.6348 - val_loss: 0.6797 - val_accuracy: 0.7419
Epoch 2/30
750/750 [==============================] - 4s 5ms/step - loss: 0.6574 - accuracy: 0.7485 - val_loss: 0.6019 - val_accuracy: 0.7655
Epoch 3/30
750/750 [==============================] - 4s 5ms/step - loss: 0.5770 - accuracy: 0.7808 - val_loss: 0.5260 - val_accuracy: 0.8071
Epoch 4/30
```

## Results:

`loss: 0.1858 - accuracy: 0.9303 - val_loss: 0.3063 - val_accuracy: 0.8982`



Try different hyperparameters to get the best result using keras tuner

```
input_shape = x_train1.shape[1:]
def model_builder(hp):
    lenet_5_model = Sequential()
    lenet_5_model.add(Conv2D(filters=hp.Int('CONV_1_FILTER',min_value=32, max_value=64),
                    kernel_size=hp.Choice('KERNEL_1_FILTER', values=[3,5]),activation='relu',  padding='same',input_shape=(32,32,1)))

    lenet_5_model.add(MaxPooling2D(strides=2))
    lenet_5_model.add(Conv2D(filters=48, kernel_size=(5,5), padding='valid', activation='relu'))
    lenet_5_model.add(Conv2D(filters=hp.Int('CONV_2_FILTER',min_value=32, max_value=128),
                    kernel_size=hp.Choice('KERNEL_2_FILTER', values=[3,5]), activation='relu',padding='valid'))


    lenet_5_model.add(MaxPooling2D(strides=2))
    lenet_5_model.add(Flatten())
    lenet_5_model.add(Dense(256, activation='relu'))
    lenet_5_model.add(Dropout(hp.Float('DROPOUT_1', min_value=0.0,max_value=0.5,default=0.25,step=0.05)))

    lenet_5_model.add(Dense(84, activation='relu'))
    lenet_5_model.add(Dense(10, activation='softmax'))
    lenet_5_model.build()
    lenet_5_model.compile(optimizer=Adam(), loss=losses.categorical_crossentropy, metrics=['accuracy'])
    lenet_5_model.compile(Adam(hp.Float('learning_rate', min_value=1e-4, max_value=1e-2, sampling='LOG')),
                   loss='categorical_crossentropy', metrics=['accuracy'])

    return lenet_5_model
```

```
[43] tuner = kt.Hyperband(model_builder,
                          objective='val_accuracy',
                          max_epochs=10,
                          directory='models',
                          project_name='mnist')
```

```
tuner.search_space_summary()
```

```
Search space summary
Default search space size: 6
CONV_1_FILTER (Int)
{'default': None, 'conditions': [], 'min_value': 32, 'max_value': 64, 'step': 1, 'sampling': 'linear'}
KERNEL_1_FILTER (Choice)
{'default': 3, 'conditions': [], 'values': [3, 5], 'ordered': True}
CONV_2_FILTER (Int)
{'default': None, 'conditions': [], 'min_value': 32, 'max_value': 128, 'step': 1, 'sampling': 'linear'}
KERNEL_2_FILTER (Choice)
{'default': 3, 'conditions': [], 'values': [3, 5], 'ordered': True}
DROPOUT_1 (Float)
{'default': 0.25, 'conditions': [], 'min_value': 0.0, 'max_value': 0.5, 'step': 0.05, 'sampling': 'linear'}
learning_rate (Float)
{'default': 0.0001, 'conditions': [], 'min_value': 0.0001, 'max_value': 0.01, 'step': None, 'sampling': 'log'}
```

```
[45] # perform early stop to prevent the model from overfitting
     stop_early = EarlyStopping(monitor='val_loss', patience=5)
     tuner.search(x_train1, train_labels_one_hot, epochs=20,
                  validation_data=(x_val1 ,val_labels_one_hot), callbacks=[stop_early])
     # Get the optimal hyperparameters
     best_hps=tuner.get_best_hyperparameters(num_trials=1)[0]
```

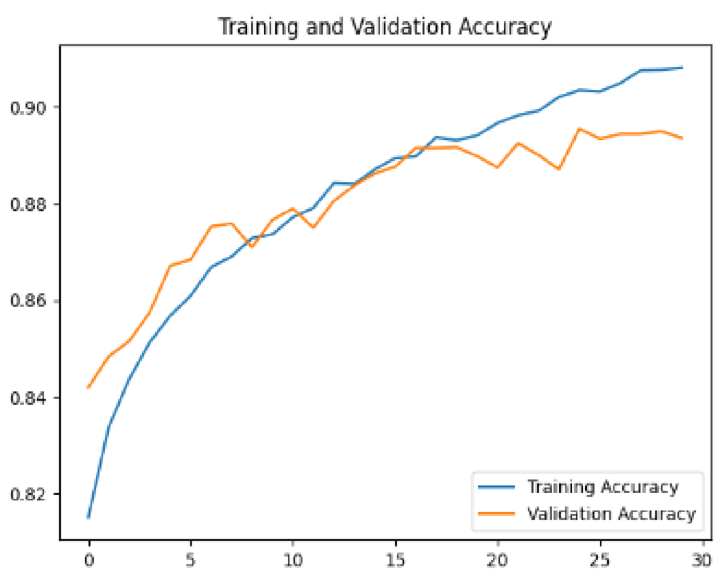
Then train our model with the best combination of hyperparameters produced by the previous code.

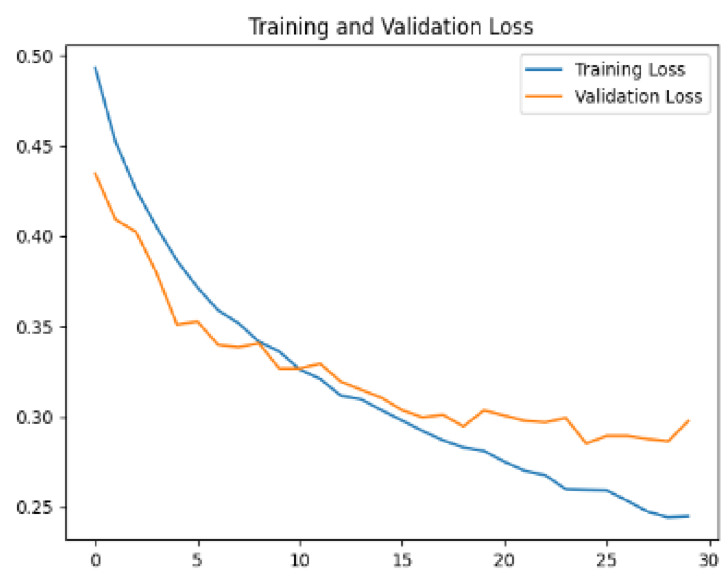
```
# Creating a Model Checkpoint to save it
filepath="New\mnist1.hdf5"
checkpoint_conv = ModelCheckpoint(filepath, monitor='val_accuracy', verbose=1, save_best_only=True, mode='max')
callbacks_list_conv = [checkpoint_conv]

import time
start_time = time.time()

mymodel = fModel.fit( x_train1, train_labels_one_hot, batch_size=64,
                    steps_per_epoch = int(np.ceil(len(x_train1)/float(64))), # Num of batches
                    epochs = 30,
                    validation_data =(x_val1, val_labels_one_hot),
                    shuffle = True,
                    callbacks=callbacks_list_conv
                          )
print(f'\nDuration: {time.time() - start_time:.0f} seconds')
```

```
750/750 [==============================] - 9s 12ms/step - loss: 0.2905 - accuracy: 0.8906 - val_loss: 0.3298 - val_accuracy: 0.8779
Epoch 17/30
742/750 [============================>.] - ETA: 0s - loss: 0.2828 - accuracy: 0.8949
Epoch 17: val_accuracy improved from 0.87792 to 0.87925, saving model to New\mnist1.hdf5
750/750 [==============================] - 5s 6ms/step - loss: 0.2826 - accuracy: 0.8950 - val_loss: 0.3250 - val_accuracy: 0.8793
Epoch 18/30
749/750 [============================>.] - ETA: 0s - loss: 0.2748 - accuracy: 0.8971
Epoch 18: val_accuracy improved from 0.87925 to 0.88617, saving model to New\mnist1.hdf5
750/750 [==============================] - 7s 10ms/step - loss: 0.2748 - accuracy: 0.8972 - val_loss: 0.3165 - val_accuracy: 0.8862
Epoch 19/30
745/750 [============================>.] - ETA: 0s - loss: 0.2637 - accuracy: 0.9020
Epoch 19: val_accuracy did not improve from 0.88617
750/750 [==============================] - 7s 10ms/step - loss: 0.2641 - accuracy: 0.9018 - val_loss: 0.3280 - val_accuracy: 0.8803
Epoch 20/30
745/750 [============================>.] - ETA: 0s - loss: 0.2569 - accuracy: 0.9042
Epoch 20: val_accuracy did not improve from 0.88617
```

## Results:

```
loss: 0.1962 - accuracy: 0.9251 - val_loss: 0.3116 - val_accuracy: 0.8940
```



Use 10-fold cross validation to evaluate the model

```
# Merge inputs and targets
inputs = np.concatenate((x_train1, x_val1), axis=0)
targets = np.concatenate((train_labels_one_hot, val_labels_one_hot), axis=0)

# Define the K-fold Cross Validator
n_folds = 10
acc_per_fold = []
loss_per_fold = []
kfold = KFold(n_splits=n_folds, shuffle=True)

stop_early = EarlyStopping(monitor='val_loss', patience=10)

# K-fold Cross Validation model evaluation
fold_no = 1
for train, test in kfold.split(inputs, targets):

    print(f'fold {fold_no} ...')
    fModel.compile(optimizer=Adam(), loss=losses.categorical_crossentropy, metrics=['accuracy'])

    # Fit data to model
    history = fModel.fit( inputs[train], targets[train], batch_size=64,
                          steps_per_epoch = int(np.ceil(len(x_train1)/float(64))),
                          epochs = 30,
                          validation_data =(inputs[test], targets[test]),
                          shuffle = True, callbacks = [stop_early])

    plot_graphs(history, type="loss")
    plot_graphs(history, type="accuracy")

    # Generate generalization metrics
    scores = fModel.evaluate(inputs[test], targets[test], verbose=0)
    acc_per_fold.append(scores[1] * 100)
    loss_per_fold.append(scores[0])

    # Increase fold number
    fold_no = fold_no + 1
```

## Results:

### For fold 1:

`loss: 0.0616 - accuracy: 0.9778 - val_loss: 0.0517 - val_accuracy: 0.9838`

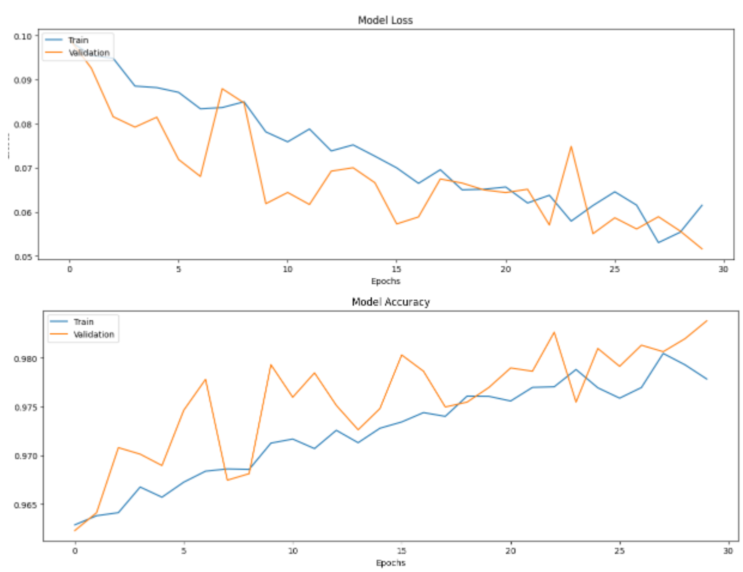



Fig.3. Fold 1 results

**For Fold 10:**

```
loss: 0.0352 - accuracy: 0.9885 - val_loss: 0.0233 - val_accuracy: 0.9925
```

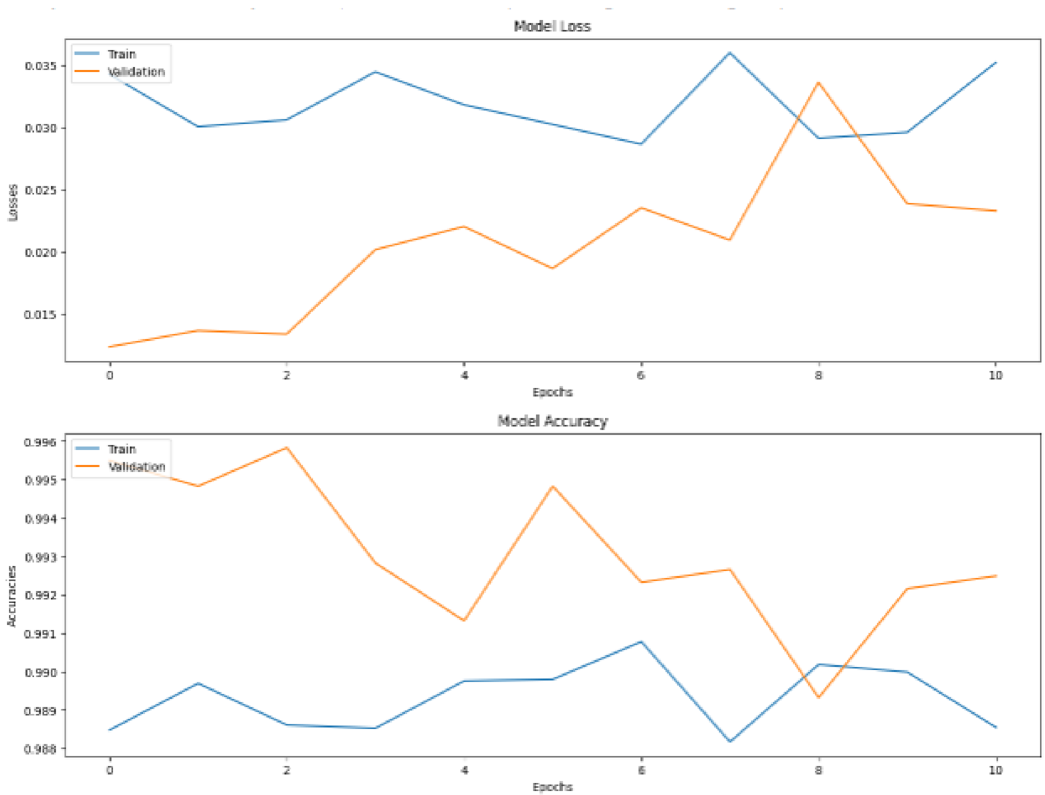



Fig.4. Fold 10 Results

**Observation:**

The range of values of accuracies is very tight across all 10 folds indicating that the model doesn't have a high variance (doesn't overfit).

## Transfer Learning

### 1) Using the pre-trained ResNet152V2 model

```python
def ResNet_builder():

    # convert the 1 dimension gray image to 3 duplicated dimensions to be able to pass them to the model
    img_input = Input(shape=(32, 32, 1))
    img_conc = tf.keras.layers.Concatenate()([img_input, img_input, img_input])

    # create the base pre-trained model
    base_model = ResNet152V2(weights='imagenet', include_top=False, input_tensor=img_conc)

    x = base_model.output
    # adding a average pooling layer
    x = GlobalMaxPooling2D()(x)
    # adding a fully-connected layer with relu activation
    # x = Dense(1024, activation='relu')(x)
    # # adding a dropout layer
    # x = Dropout(0.4)(x)
    # adding a fully-connected layer with relu activation
    x = Dense(512, activation='relu')(x)
    # adding a dropout layer
    x = Dropout(0.4)(x)
    # adding a fully-connected layer with relu activation
    x = Dense(128, activation='relu')(x)
    # adding the output layer which has 10 classes with a softmax
    predictions = Dense(10, activation='softmax')(x)

    # this is the model we will train
    model = Model(inputs=base_model.input, outputs=predictions)

    # first: train only the top layers (which were randomly initialized)
    # i.e. freeze all convolutional InceptionV3 layers
    for layer in base_model.layers:
        layer.trainable = False

    model.compile(optimizer=Adam(learning_rate=0.001),
                  loss=CategoricalCrossentropy(),
                  metrics=['accuracy'])

    return model
```

```python
resnet_model = ResNet_builder()
```

## Results:

`loss: 0.8677 - accuracy: 0.6735 - val_loss: 0.7336 - val_accuracy: 0.7231`

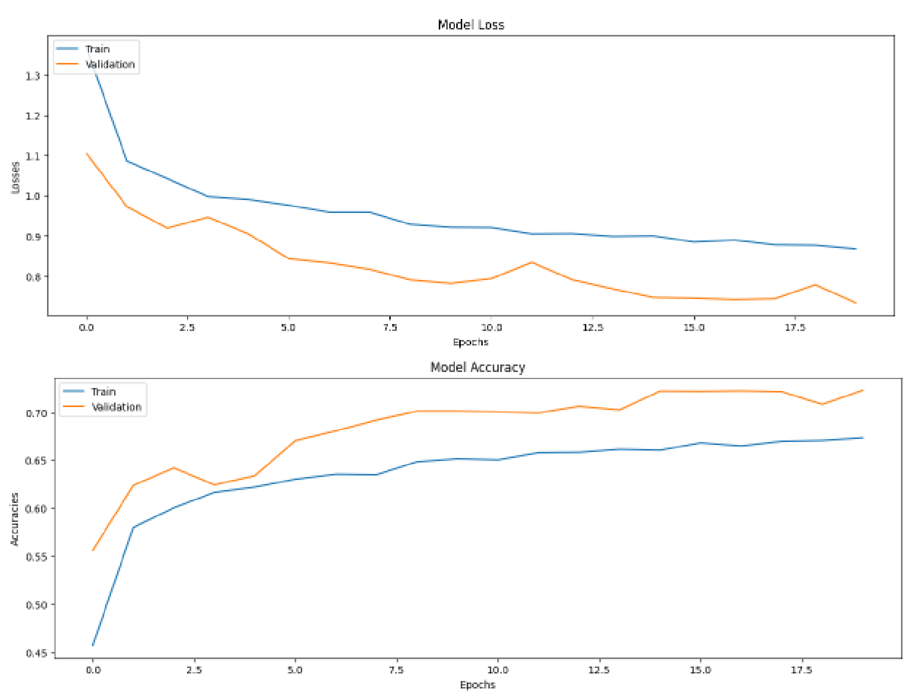



Fig.4. ResNet152V2 accuracy and loss curves

## 2) Using the pretrained EfficientNetB1 model

```
def EfficientNetB1_builder():

    # convert the 1 dimension gray image to 3 duplicated dimensions to be able to pass them to the model
    img_input = Input(shape=(32, 32, 1))
    img_conc = tf.keras.layers.Concatenate()([img_input, img_input, img_input])

    # create the base pre-trained model
    base_model = EfficientNetB1(weights='imagenet', include_top=False, input_tensor=img_conc)

    x = base_model.output
    # adding a average pooling layer
    x = GlobalMaxPooling2D()(x)
    # adding a fully-connected layer with relu activation
    # x = Dense(1024, activation='relu')(x)
    # # adding a dropout layer
    # x = Dropout(0.4)(x)
    # adding a fully-connected layer with relu activation
    x = Dense(512, activation='relu')(x)
    # adding a dropout layer
    x = Dropout(0.4)(x)
    # adding a fully-connected layer with relu activation
    x = Dense(128, activation='relu')(x)
    # adding the output layer which has 10 classes with a softmax
    predictions = Dense(10, activation='softmax')(x)

    # this is the model we will train
    model = Model(inputs=base_model.input, outputs=predictions)

    # first: train only the top layers (which were randomly initialized)
    # i.e. freeze all convolutional InceptionV3 layers
    for layer in base_model.layers:
        layer.trainable = False

    model.compile(optimizer=Adam(learning_rate=0.001),
                  loss=CategoricalCrossentropy(),
                  metrics=['accuracy'])

    return model
```

```
EfficientNetB1_model = EfficientNetB1_builder()
```

## Results:

`loss: 2.3028 - accuracy: 0.0973 - val_loss: 2.3026 - val_accuracy: 0.1001`

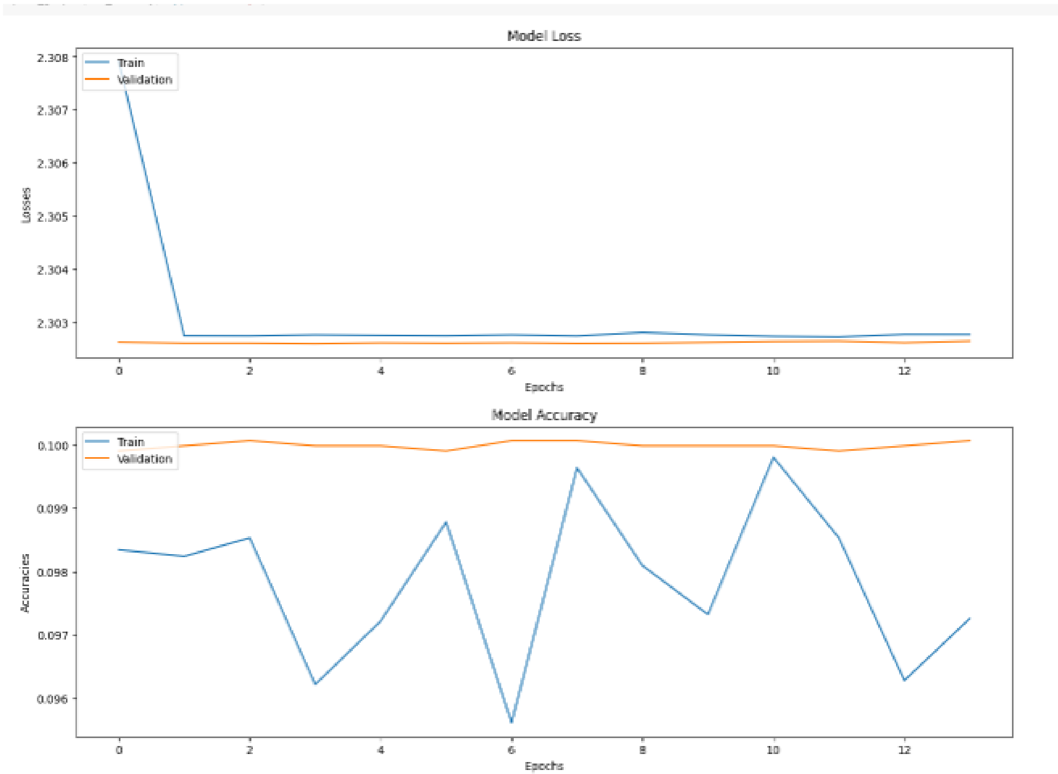



Fig.4. EfficientNetB1 accuracy and loss curves

## 3) Using the pretrained VGG16 model

```
def VGG16_builder():

    # convert the 1 dimension gray image to 3 duplicated dimensions to be able to pass them to the model
    img_input = Input(shape=(32, 32, 1))
    img_conc = tf.keras.layers.Concatenate()([img_input, img_input, img_input])

    # create the base pre-trained model
    base_model = VGG16(weights='imagenet', include_top=False, input_tensor=img_conc)

    # first: train only the top layers (which were randomly initialized)
    # i.e. freeze all convolutional InceptionV3 layers
    for layer in base_model.layers:
        layer.trainable = False

    x = base_model.output
    # adding a average pooling layer
    x = GlobalMaxPooling2D()(x)
    # adding a fully-connected layer with relu activation
    x = Dense(1024, activation='relu')(x)
    # # adding a dropout layer
    x = Dropout(0.4)(x)
    # adding a fully-connected layer with relu activation
    x = Dense(512, activation='relu')(x)
    # adding a dropout layer
    x = Dropout(0.4)(x)
    # adding a fully-connected layer with relu activation
    x = Dense(128, activation='relu')(x)
    # adding the output layer which has 10 classes with a softmax
    predictions = Dense(10, activation='softmax')(x)

    # this is the model we will train
    model = Model(inputs=base_model.input, outputs=predictions)

    model.compile(optimizer=Adam(learning_rate=0.001),
                  loss=CategoricalCrossentropy(),
                  metrics=['accuracy'])

    return model
```

## Results:

**loss: 2.3028 - accuracy: 0.0985 - val_loss: 2.3026 - val_accuracy: 0.1000**

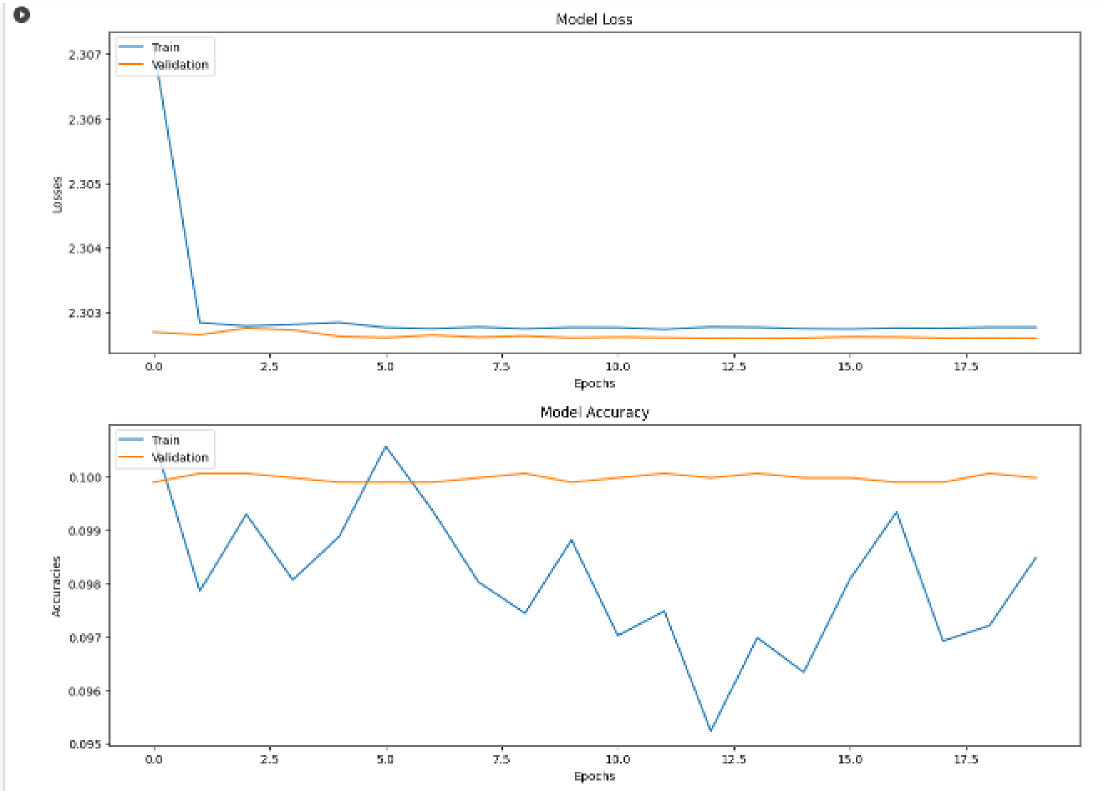



Fig.4. VGG16 accuracy and loss curves

## Conclusion:

- The accuracy dropped when we used transfer learning especially when using EfficientNetB1 and VGG16 models. Transfer learning may fail if there is a domain mismatch between the dataset for pretext tasks and the dataset for the downstream task.

- Although the pre-trained models may converge, they will remain locked at a local minimum. As a result, the performance will be no better than if you started from scratch.