nt-1-group-62-fashion-mnist-v-0-01

June 24, 2023

0.1 Group No : 62

0.2 Group Member Names:

1. Avishek Kumar (2021sc04238@wilp.bits-pilani.ac.in)
2. Soumyadipta Maiti (2021sc04237@wilp.bits-pilani.ac.in)
3. Kumar Sreyam Nandi (2021sc04701@wilp.bits-pilani.ac.in)
4. Tanish Khanna (2021sc04236@wilp.bits-pilani.ac.in)

1 1. Import the required libraries

[391]:
```
import numpy as np
import pandas as pd
from matplotlib import pyplot
import matplotlib.pyplot as plt
import seaborn as sns

#for download & unzip of files
import zipfile, requests
import urllib.request

#for modeling of Artificial Neural Network
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Dropout, Flatten
from tensorflow.keras.optimizers import SGD, Adam, RMSprop
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras import regularizers
from tensorflow.keras.utils import to_categorical

#for cross-validation
from sklearn.model_selection import StratifiedKFold , KFold ,RepeatedKFold,␣
 ↪cross_val_score

#for keras tuner
import keras_tuner
from keras_tuner.tuners import RandomSearch
from keras_tuner.engine.hyperparameters import HyperParameters
```

```
# for Evaluation of Classification
from sklearn.metrics import classification_report, confusion_matrix

import time
import warnings
warnings.filterwarnings(action = 'ignore')
```

2 2. Data Acquisition – Score: 0.5 Mark

For the problem identified by you, students have to find the data source themselves from any data source.

Fashion_mnist dataset

Fashion-MNIST is a dataset of Zalando's article images consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes.

More details can be found at fashion_mnist homepage

2.1 2.1 Code for converting the above downloaded data into a form suitable for DL

Below Code will download file from requested url into specified filename. But due to huge size of file, we are disabling calling of downlaod function

[392]:
```
def download(url, name):
    r = requests.get(url, allow_redirects=True)
    open(name,'wb').write(r.content)
```

[393]:
```
# download(url='https://www.cs.toronto.edu/%7Ekriz/cifar-10-python.tar.gz', 
 ↪name='cifar-10-python.tar.gz')
```

Below code will unzip .tar file. It's commented due to huge size of file.

[394]:
```
# import tarfile
# file = tarfile.open('cifar-10-python.tar.gz')
# file.extractall('./Destination_FolderName')
# file.close()
```

[395]:
```
def unpickle(filecontent):
    import pickle
    with open(filecontent, 'rb') as file:
        dictionary = pickle.load(file, encoding='bytes')
    return dictionary
```

[396]:
```
# file = 'Destination_FolderName/cifar-10-batches-py/data_batch_1'
# whole_file = unpickle(file)
# print(whole_file)
```

Alternatively, this dataset can be imported directly from TensorFlow Datasets

[397]:
```
from tensorflow.keras.datasets import fashion_mnist
```

Loading of Fashion-MNIST dataset

This is a dataset of 60,000 28x28 grayscale images of 10 fashion categories, along with a test set of 10,000 images. This dataset can be used as a drop-in replacement for MNIST.

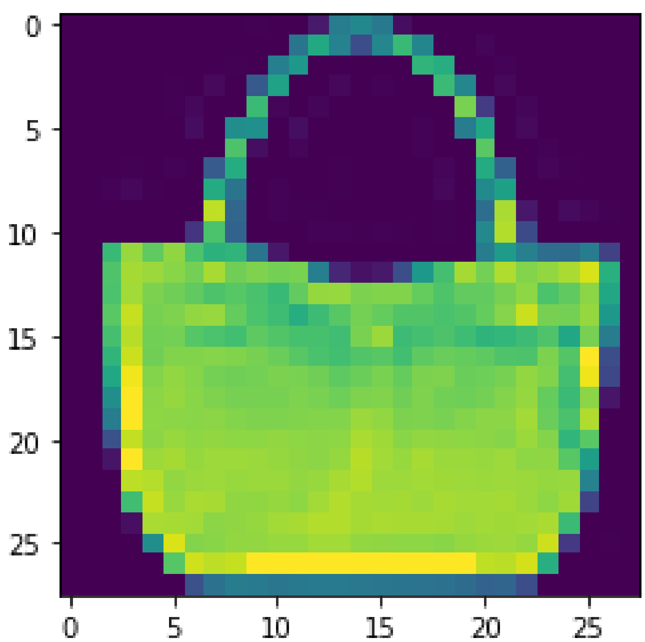
[398]:
```
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

y_train_org = y_train.copy()
y_test_org = y_test.copy()
```

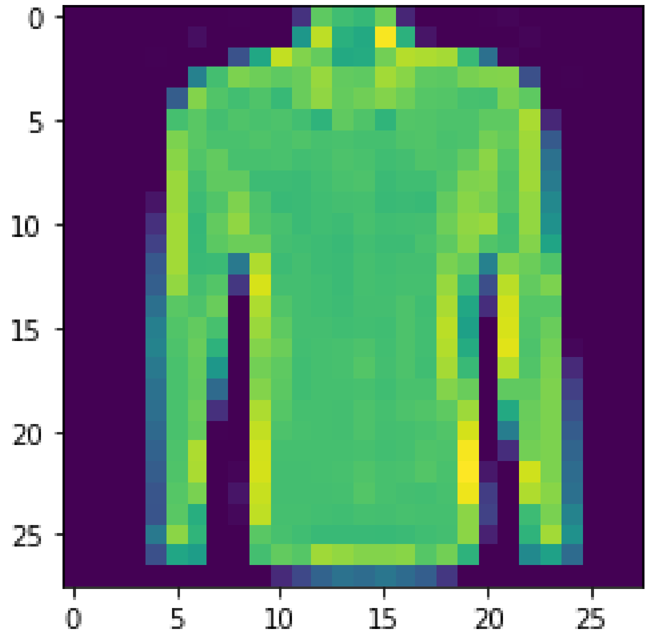
Image View from dataset

[399]:
```
def image_view(image_row):
    plt.figure(figsize=(4,4))
    plt.imshow(x_train[image_row])
    plt.show()
```

[400]:
```
image_view(image_row=100)
```



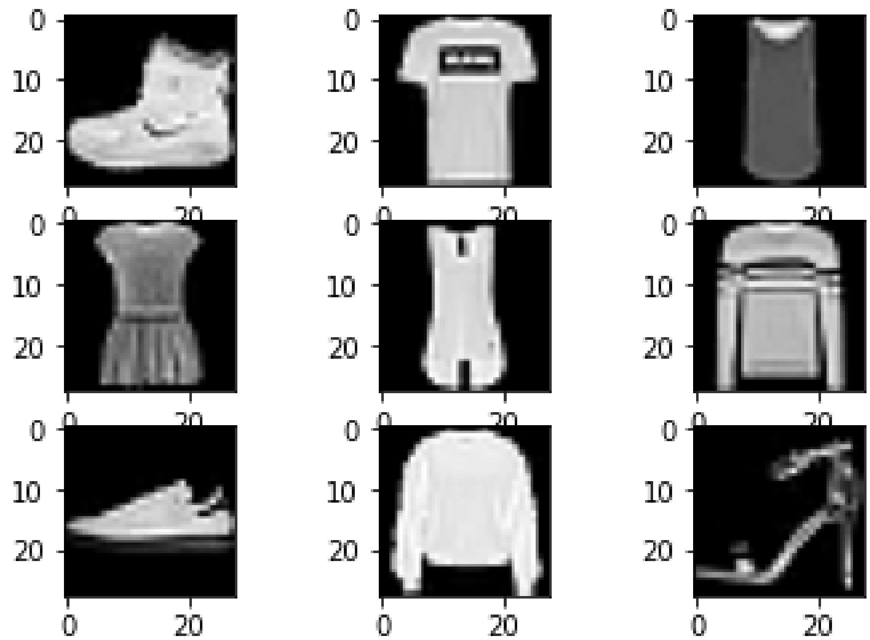
[401]:
```
image_view(image_row=40)
```



ploting of first few images in grayscale

[402]:
```
for count in range(9):
 # define subplot
 pyplot.subplot(330 + 1 + count)
 # plot raw pixel data
 pyplot.imshow(x_train[count], cmap=pyplot.get_cmap('gray'))

# show the figure
pyplot.show()
```



[403]:
```
plt.figure(figsize=(10,10))

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']


for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_train[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[y_train[i]])
plt.show()
```

2.2 2.1 Write your observations from the above.

1. Size of the dataset
2. What type of data attributes are there?
3. What are you classifying?
4. Plot the distribution of the categories of the target / label.

Size of Dataset

[404]:
```
print(f'Size of Entire FASHION_MNIST DatatSet : {x_train.shape[0]+x_test.
  ↪shape[0]},{x_train.shape[1:]}')
print(f'Size of Training DatatSet : {x_train.shape}')
print(f'Size of Testing DatatSet : {x_test.shape}')
```

```
Size of Entire FASHION_MNIST DatatSet : 70000,(28, 28)
Size of Training DatatSet : (60000, 28, 28)
Size of Testing DatatSet : (10000, 28, 28)
```

x_train: uint8 NumPy array of grayscale image data with shapes `(60000, 28, 28)`, containing the training data.

y_train: uint8 NumPy array of labels (integers in range 0-9) with shape `(60000,)` for the training data.

x_test: uint8 NumPy array of grayscale image data with shapes (10000, 28, 28), containing the test data.

y_test: uint8 NumPy array of labels (integers in range 0-9) with shape `(10000,)` for the test data.

Type of Data Attributes

[405]:
```python
print(f'Data Type of Fields : {x_train.dtype}')
```

```
Data Type of Fields : uint8
```

What are you classifying?

[406]:
```python
print(f' Unique values to be classified : {np.unique(y_train)}')
```

```
 Unique values to be classified : [0 1 2 3 4 5 6 7 8 9]
```

The classes are:

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

Plot Distribution of Categories of Target/Label

[407]:
```python
unique_plt, count_plt = np.unique(y_train, return_counts=True)
```

[408]:
```python
plt.bar(unique_plt, count_plt)
plt.xlabel('Unique Target Values')
plt.ylabel('Counts')
plt.title('Counts of Unique Target Values of Training Data')
plt.show()
```

3 3. Data Preparation – Score: 1 Mark

Perform the data prepracessing that is required for the data that you have downloaded.

This stage depends on the dataset that is used.

3.1 3.1 Apply pre-processing techiniques

- to remove duplicate data
- to impute or remove missing data
- to remove data inconsistencies
- Encode categorical data
- Normalize the data
- Feature Engineering
- Stop word removal, lemmatiation, stemming, vectorization

IF ANY

Removal of Duplicate Data

[409]:
```
unique,count = np.unique(x_train, axis=0, return_counts=True)
duplicate = unique[count>1]
```

[410]:
```
print(f'Number of Duplicate Rows in Training Data : {duplicate.shape[0]}')
```

```
Number of Duplicate Rows in Training Data : 0
```

Imputation or Removal of Missing Data

[411]:
```
print(f'Number of Missing Rows in Training Data : {np.isnan(x_train).sum()}')
```

```
Number of Missing Rows in Training Data : 0
```

Removal of Data Inconsistencies

[412]:
```
print(f'Training Data set is having minimum value as {x_train.min()} & maximum␣
↪value as {x_train.max()}')
print(f'Testing Data set is having minimum value as {x_test.min()} & maximum␣
↪value as {x_test.max()}')
```

```
Training Data set is having minimum value as 0 & maximum value as 255
Testing Data set is having minimum value as 0 & maximum value as 255
```

So, All values lie between 0 to 255. Hence, all values are consistent wrt this data.

Encoding of Categorical Data

[413]:
```
print(f' There is no categorical values as datatype of its values are {x_train.
↪dtype}')
```

```
 There is no categorical values as datatype of its values are uint8
```

Normalization of Data

[414]:
```
print(f'Training Data set is having minimum value as {x_train.min()} & maximum␣
↪value as {x_train.max()}')
```

```
Training Data set is having minimum value as 0 & maximum value as 255
```

[415]:
```
#So, we need to normalize training & test data on deviding by 255 so that all␣
↪values lie between 0 & 1
x_train = x_train/255
x_test = x_test/255
```

Feature Engineering

[416]:
```
#Reshaping Input Variables
x_train = x_train.reshape(x_train.shape[0], x_train.shape[1], x_train.shape[2],␣
↪1)
x_test = x_test.reshape(x_test.shape[0], x_test.shape[1], x_test.shape[2], 1)
```

[417]:
```
#conversion of input values from Integer to Float for more accuracy
x_train = x_train.astype(float)
x_test = x_test.astype(float)
```

No extra feature Engineering is required except Normalization as mentioned above

Text Preprocessing

Stop word removal, lemmatiation, stemming, vectorization are not Applicable as it's not a Natural language Problem

[ ]:

```

```

3.2 3.2 Identify the target variables.

- Separate the data front the target such that the dataset is in the form of (X,y) or (Features, Label)
- Discretize / Encode the target variable or perform one-hot encoding on the target or any other as and if required.

Separation of Data for Independent and Target Variable(X,y)

Seperation of Data into Indepedent(x_train, x_test) & Target Variable(y_train, y_test) is already performed previously while calling **load_data()** method as shown below:

(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

[418]:

```
print(f'Shape of Indepedent Variables = {x_train.shape} & Target Variable =␣
↪{y_train.shape} in Training Data')
```

```
Shape of Indepedent Variables = (60000, 28, 28, 1) & Target Variable = (60000,)
in Training Data
```

One-hot Encoding of Target variable

[419]:

```
y_train_cat = to_categorical(y_train,num_classes=10)
y_test_cat = to_categorical(y_test,num_classes=10)
```

[420]:

```
print(f'Shape of Target Variable in Training Set before & after One-hot␣
↪Encoding are {y_train.shape} & {y_train_cat.shape}')
print(f'Shape of Target Variable in Testing Set before & after One-hot Encoding␣
↪are {y_test.shape} & {y_test_cat.shape}')
```

```
Shape of Target Variable in Training Set before & after One-hot Encoding are
(60000,) & (60000, 10)
Shape of Target Variable in Testing Set before & after One-hot Encoding are
(10000,) & (10000, 10)
```

3.3 3.3 Split the data into training set and testing set

Split of Data into Training(x_train, y_train) & Testing Set(x_test, y_test) is already performed previously while calling **load_data()** method as shown below:

(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

[421]:

```
print(f'Shape of Training Set = {x_train.shape} & Testing Set = {x_test.shape}␣
↪for independent variables')
```

```
Shape of Training Set = (60000, 28, 28, 1) & Testing Set = (10000, 28, 28, 1)
for independent variables
```

3.4 3.4 Preprocessing report

Mention the method adopted and justify why the method was used * to remove duplicate data, if present * to impute or remove missing data, if present * to remove data inconsistencies, if present * to encode categorical data * the normalization technique used

If the any of the above are not present, then also add in the report below.

Report the size of the training dataset and testing dataset

Methods adopted and justify why the method was used

- to remove duplicate data, if present : **unique,count = Numpy.unique(x_train, axis=0, return_counts=True) count** is used to return counts of all unique rows of training arrayset & then checking their counts > 1 to detect duplicacy. Above method automatically removes duplicate values & store unique values in **unique** variable.
- to impute or remove missing data, if present: No missing values are present in this dataset.
- to remove data inconsistencies, if present: No inconsistent data are present in this dataset.
- to encode categorical data: There is no categorical values as datatype of its values are uint8. One-hot Encoding of Target variable is performed to convert into from 1 to 10.
- the normalization technique used: Training Data set is having minimum value as 0.0 & maximum value as 255.0. So, we need to normalize training & test data on deviding by 255 so that all values lie between 0 & 1

[ ]:

4 4. Deep Neural Network Architecture - Score: Marks

4.1 4.1 Design the architecture that you will be using

- Sequential Model Building with Activation for each layer.
- Add dense layers, specifying the number of units in each layer and the activation function used in the layer.
- Use Relu Activation function in each hidden layer
- Use Sigmoid / softmax Activation function in the output layer as required

DO NOT USE CNN OR RNN.

Modeling using Artificial neural Network (ANN)

Although this Computer Vision problem could be solved preferebly using Convolution Neural Network (CNN), we could not use it as mentioned in problem statement

Creation of ANN Model

Sequential Model Building with Activation Function

Configuration of Each Layer (no of units, activation function, etc)

[422]:
```
def ANN_Creation(x_train, y_train_cat, no_hidden_layer=5, kernel_regularizer=0.
↪01, bias_regularizer=0.01,
                 activity_regularizer=0.01,dropout_rate=0.2):

    model_ann = Sequential()

    #Input layer
    model_ann.add(Flatten(input_shape=(x_train.shape[1], x_train.shape[2],␣
↪x_train.shape[3])))
    model_ann.add(Dropout(dropout_rate))

    for count in range(no_hidden_layer, 0, -1):

        #Hidden Layer i
        model_ann.add(Dense(units=count*32, activation='relu',␣
↪kernel_regularizer=regularizers.l2(kernel_regularizer),
                            bias_regularizer=regularizers.l2(bias_regularizer),
                            activity_regularizer=regularizers.
↪l2(activity_regularizer)))

        model_ann.add(Dropout(dropout_rate))

    #Output Layer
    model_ann.add(Dense(units=y_train_cat.shape[1], activation='softmax'))
    return model_ann
```

[423]:
```
model_ann = ANN_Creation(x_train=x_train, y_train_cat=y_train_cat,␣
↪no_hidden_layer=5, dropout_rate=0.0)
```

4.2 4.2 DNN Report

Report the following and provide justification for the same.

- Number of layers
- Number of units in each layer
- Total number of trainable parameters

[424]:
```
model_ann.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_1 (Flatten)         (None, 784)               0

 dropout (Dropout)           (None, 784)               0

 dense_6 (Dense)             (None, 160)               125600
```

```
 dropout_1 (Dropout)         (None, 160)               0

 dense_7 (Dense)             (None, 128)               20608

 dropout_2 (Dropout)         (None, 128)               0

 dense_8 (Dense)             (None, 96)                12384

 dropout_3 (Dropout)         (None, 96)                0

 dense_9 (Dense)             (None, 64)                6208

 dropout_4 (Dropout)         (None, 64)                0

 dense_10 (Dense)            (None, 32)                2080

 dropout_5 (Dropout)         (None, 32)                0

 dense_11 (Dense)            (None, 10)                330

=================================================================
Total params: 167,210
Trainable params: 167,210
Non-trainable params: 0
_________________________________________________________________
```

Summary of Model

- Number of layers: As it's a Deep Learning Model, we need to set minimum total no of Layers = 4 having 3 hidden layers & one output Layer. Input Layer is not considered while calculating total no of Layers of any Neural Network. Will keep on increasing hidden layer to check its performance.
- Number of units in each layer: We keep this as multiple of 32. e.g.–> If total no of hidden layers are 5, then 1-st till 5-th hidden layers are having number of units/neurons as n * 32 in decreasing order so that they are 160(=5*32), 128(=4*32), 96(=3*32), 64(=5*32) & 32(=1*32) respectively.

  No. of units of First Layer = Flattening of Input Demensions of Training Set = 28*28 = 784
  No. of units of Output Layer = Number of Classes of Multi-Class Target variable = 10
- Total number of trainable parameters: It's 167,210 (summing 0, 125600, 20608, 12384, 6208, 2080 & 330 respectively starting from input till output layer).

[ ]:

5 5. Training the model - Score: 1 Mark

5.1 5.1 Configure the training

Configure the model for training, by using appropriate optimizers and regularizations

Compile with categorical CE loss and metric accuracy.

Optimizer

- Optimizers are algorithms or methods used to change attributes (such as weights, learning rate, momentum) of neural network in order to reduce losses.

[425]:
```
adam = Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07,␣
↪amsgrad=False,)
```

Regularization

It's used to prevent overfitting and improve the generalization performance of a model. It involves adding a penalty term to the loss function during training. Regularization methods include L1 and L2 regularization, dropout, early stopping, and more.

Compilation of ANN

- loss='categorical_crossentropy' is used for calcuation of loss for multi-class categorical variable
- metrics: List of metrics('accuracy', 'mse') to be evaluated by the model during training & testing. 'accuracy' is used here.

[426]:
```
model_ann.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
```

Training of simple ANN Model as defined in section 4.1

[427]:
```
early_stop = EarlyStopping(monitor='val_loss', patience=3)
```

[428]:
```
print(f'{x_train.shape} ; {y_train_cat.shape} :: {x_test.shape}; {y_test_cat.
↪shape}')
```

```
(60000, 28, 28, 1) ; (60000, 10) :: (10000, 28, 28, 1); (10000, 10)
```

[429]:
```
model_ann.fit(x=x_train,
              y=y_train_cat,
              epochs=100,
              batch_size=200,
              validation_data=(x_test, y_test_cat),
              verbose=2,
              callbacks=[early_stop]
             )
```

```
Epoch 1/100
300/300 - 6s - loss: 2.5956 - accuracy: 0.6258 - val_loss: 1.2807 -
val_accuracy: 0.8027 - 6s/epoch - 21ms/step
Epoch 2/100
300/300 - 2s - loss: 1.1574 - accuracy: 0.8240 - val_loss: 1.0816 -
val_accuracy: 0.8282 - 2s/epoch - 6ms/step
Epoch 3/100
300/300 - 2s - loss: 1.0171 - accuracy: 0.8446 - val_loss: 0.9991 -
val_accuracy: 0.8384 - 2s/epoch - 5ms/step
Epoch 4/100
300/300 - 1s - loss: 0.9484 - accuracy: 0.8508 - val_loss: 0.9539 -
val_accuracy: 0.8388 - 1s/epoch - 4ms/step
Epoch 5/100
300/300 - 1s - loss: 0.9036 - accuracy: 0.8540 - val_loss: 0.9183 -
val_accuracy: 0.8410 - 1s/epoch - 4ms/step
Epoch 6/100
300/300 - 1s - loss: 0.8756 - accuracy: 0.8549 - val_loss: 0.8986 -
val_accuracy: 0.8433 - 1s/epoch - 5ms/step
Epoch 7/100
300/300 - 1s - loss: 0.8513 - accuracy: 0.8565 - val_loss: 0.8592 -
val_accuracy: 0.8505 - 1s/epoch - 5ms/step
Epoch 8/100
300/300 - 2s - loss: 0.8328 - accuracy: 0.8572 - val_loss: 0.8643 -
val_accuracy: 0.8461 - 2s/epoch - 5ms/step
Epoch 9/100
300/300 - 2s - loss: 0.8136 - accuracy: 0.8593 - val_loss: 0.8447 -
val_accuracy: 0.8411 - 2s/epoch - 6ms/step
Epoch 10/100
300/300 - 2s - loss: 0.8052 - accuracy: 0.8580 - val_loss: 0.8158 -
val_accuracy: 0.8517 - 2s/epoch - 6ms/step
Epoch 11/100
300/300 - 1s - loss: 0.7856 - accuracy: 0.8601 - val_loss: 0.8199 -
val_accuracy: 0.8478 - 1s/epoch - 5ms/step
Epoch 12/100
300/300 - 1s - loss: 0.7732 - accuracy: 0.8617 - val_loss: 0.7994 -
val_accuracy: 0.8477 - 1s/epoch - 4ms/step
Epoch 13/100
300/300 - 1s - loss: 0.7638 - accuracy: 0.8619 - val_loss: 0.7977 -
val_accuracy: 0.8526 - 1s/epoch - 4ms/step
Epoch 14/100
300/300 - 1s - loss: 0.7524 - accuracy: 0.8628 - val_loss: 0.7925 -
val_accuracy: 0.8482 - 1s/epoch - 4ms/step
Epoch 15/100
300/300 - 1s - loss: 0.7478 - accuracy: 0.8611 - val_loss: 0.7763 -
val_accuracy: 0.8478 - 1s/epoch - 4ms/step
Epoch 16/100
300/300 - 1s - loss: 0.7419 - accuracy: 0.8610 - val_loss: 0.7676 -
val_accuracy: 0.8495 - 1s/epoch - 4ms/step
```

```
Epoch 17/100
300/300 - 1s - loss: 0.7305 - accuracy: 0.8627 - val_loss: 0.7697 -
val_accuracy: 0.8513 - 1s/epoch - 4ms/step
Epoch 18/100
300/300 - 1s - loss: 0.7264 - accuracy: 0.8629 - val_loss: 0.7578 -
val_accuracy: 0.8481 - 1s/epoch - 4ms/step
Epoch 19/100
300/300 - 1s - loss: 0.7174 - accuracy: 0.8638 - val_loss: 0.7535 -
val_accuracy: 0.8477 - 1s/epoch - 4ms/step
Epoch 20/100
300/300 - 1s - loss: 0.7120 - accuracy: 0.8630 - val_loss: 0.7479 -
val_accuracy: 0.8458 - 1s/epoch - 5ms/step
Epoch 21/100
300/300 - 2s - loss: 0.7119 - accuracy: 0.8615 - val_loss: 0.7686 -
val_accuracy: 0.8377 - 2s/epoch - 5ms/step
Epoch 22/100
300/300 - 2s - loss: 0.7039 - accuracy: 0.8623 - val_loss: 0.7506 -
val_accuracy: 0.8489 - 2s/epoch - 5ms/step
Epoch 23/100
300/300 - 1s - loss: 0.6968 - accuracy: 0.8648 - val_loss: 0.7436 -
val_accuracy: 0.8459 - 1s/epoch - 5ms/step
Epoch 24/100
300/300 - 1s - loss: 0.6946 - accuracy: 0.8630 - val_loss: 0.7302 -
val_accuracy: 0.8551 - 1s/epoch - 4ms/step
Epoch 25/100
300/300 - 1s - loss: 0.6941 - accuracy: 0.8617 - val_loss: 0.7356 -
val_accuracy: 0.8470 - 1s/epoch - 4ms/step
Epoch 26/100
300/300 - 1s - loss: 0.6838 - accuracy: 0.8639 - val_loss: 0.7073 -
val_accuracy: 0.8592 - 1s/epoch - 4ms/step
Epoch 27/100
300/300 - 1s - loss: 0.6794 - accuracy: 0.8653 - val_loss: 0.7064 -
val_accuracy: 0.8569 - 1s/epoch - 4ms/step
Epoch 28/100
300/300 - 1s - loss: 0.6807 - accuracy: 0.8635 - val_loss: 0.7299 -
val_accuracy: 0.8440 - 1s/epoch - 4ms/step
Epoch 29/100
300/300 - 1s - loss: 0.6754 - accuracy: 0.8639 - val_loss: 0.7006 -
val_accuracy: 0.8569 - 1s/epoch - 4ms/step
Epoch 30/100
300/300 - 1s - loss: 0.6746 - accuracy: 0.8630 - val_loss: 0.7206 -
val_accuracy: 0.8451 - 1s/epoch - 4ms/step
Epoch 31/100
300/300 - 1s - loss: 0.6718 - accuracy: 0.8644 - val_loss: 0.7127 -
val_accuracy: 0.8512 - 1s/epoch - 4ms/step
Epoch 32/100
300/300 - 1s - loss: 0.6646 - accuracy: 0.8645 - val_loss: 0.7001 -
val_accuracy: 0.8489 - 1s/epoch - 5ms/step
```

```
Epoch 33/100
300/300 - 2s - loss: 0.6574 - accuracy: 0.8665 - val_loss: 0.7029 -
val_accuracy: 0.8489 - 2s/epoch - 5ms/step
Epoch 34/100
300/300 - 2s - loss: 0.6587 - accuracy: 0.8653 - val_loss: 0.6982 -
val_accuracy: 0.8486 - 2s/epoch - 5ms/step
Epoch 35/100
300/300 - 1s - loss: 0.6589 - accuracy: 0.8651 - val_loss: 0.7021 -
val_accuracy: 0.8491 - 1s/epoch - 5ms/step
Epoch 36/100
300/300 - 1s - loss: 0.6526 - accuracy: 0.8665 - val_loss: 0.6849 -
val_accuracy: 0.8503 - 1s/epoch - 4ms/step
Epoch 37/100
300/300 - 1s - loss: 0.6501 - accuracy: 0.8656 - val_loss: 0.7020 -
val_accuracy: 0.8482 - 1s/epoch - 4ms/step
Epoch 38/100
300/300 - 1s - loss: 0.6502 - accuracy: 0.8655 - val_loss: 0.6847 -
val_accuracy: 0.8533 - 1s/epoch - 4ms/step
Epoch 39/100
300/300 - 1s - loss: 0.6483 - accuracy: 0.8665 - val_loss: 0.6816 -
val_accuracy: 0.8530 - 1s/epoch - 4ms/step
Epoch 40/100
300/300 - 1s - loss: 0.6470 - accuracy: 0.8643 - val_loss: 0.6856 -
val_accuracy: 0.8508 - 1s/epoch - 4ms/step
Epoch 41/100
300/300 - 1s - loss: 0.6389 - accuracy: 0.8693 - val_loss: 0.7020 -
val_accuracy: 0.8464 - 1s/epoch - 4ms/step
Epoch 42/100
300/300 - 1s - loss: 0.6421 - accuracy: 0.8642 - val_loss: 0.6737 -
val_accuracy: 0.8504 - 1s/epoch - 4ms/step
Epoch 43/100
300/300 - 1s - loss: 0.6388 - accuracy: 0.8658 - val_loss: 0.6673 -
val_accuracy: 0.8531 - 1s/epoch - 4ms/step
Epoch 44/100
300/300 - 1s - loss: 0.6389 - accuracy: 0.8656 - val_loss: 0.6808 -
val_accuracy: 0.8462 - 1s/epoch - 5ms/step
Epoch 45/100
300/300 - 2s - loss: 0.6348 - accuracy: 0.8660 - val_loss: 0.6597 -
val_accuracy: 0.8579 - 2s/epoch - 5ms/step
Epoch 46/100
300/300 - 2s - loss: 0.6338 - accuracy: 0.8668 - val_loss: 0.6625 -
val_accuracy: 0.8575 - 2s/epoch - 5ms/step
Epoch 47/100
300/300 - 1s - loss: 0.6284 - accuracy: 0.8676 - val_loss: 0.6592 -
val_accuracy: 0.8542 - 1s/epoch - 4ms/step
Epoch 48/100
300/300 - 1s - loss: 0.6295 - accuracy: 0.8673 - val_loss: 0.6952 -
val_accuracy: 0.8405 - 1s/epoch - 4ms/step
```

```
Epoch 49/100
300/300 - 1s - loss: 0.6309 - accuracy: 0.8661 - val_loss: 0.6744 -
val_accuracy: 0.8535 - 1s/epoch - 4ms/step
Epoch 50/100
300/300 - 1s - loss: 0.6281 - accuracy: 0.8677 - val_loss: 0.6700 -
val_accuracy: 0.8572 - 1s/epoch - 4ms/step
```

[429]:
```
<keras.callbacks.History at 0x1c1612d37c0>
```

[430]:
```
loss = pd.DataFrame(model_ann.history.history)
```

5.2 5.2 Train the model

Train Model with cross validation, with total time taken shown for 20 epochs.

Use SGD.

ANN Model Training with Cross Validation

[431]:
```
# Average loss and Accuracy Measures for Cross-Validation
def average_loss_accuracy(losses, accuracies, val_losses, val_accuracies):

    ls_im = pd.DataFrame(losses)
    ls_im = ls_im.transpose()
    ls_im = ls_im.mean(axis=1)

    ac_im = pd.DataFrame(accuracies)
    ac_im = ac_im.transpose()
    ac_im = ac_im.mean(axis=1)

    val_ls_im = pd.DataFrame(val_losses)
    val_ls_im = val_ls_im.transpose()
    val_ls_im = val_ls_im.mean(axis=1)

    val_ac_im = pd.DataFrame(val_accuracies)
    val_ac_im = val_ac_im.transpose()
    val_ac_im = val_ac_im.mean(axis=1)

    loss_im = {'loss': ls_im, 'accuracy': ac_im, 'val_loss': val_ls_im,
 ↪'val_accuracy': val_ac_im}
    loss_cv = pd.DataFrame(data=loss_im)
    return loss_cv
```

[432]:
```
def ann_with_cross_val(x_train, y_train, no_hiddenlayer=5, dropout_rate=0.2,
 ↪k_fold=5, optimizers ='adam',
                       loss_val='categorical_crossentropy',
 ↪metrics_val='accuracy', display_time = 'X'):
```

```
    # Create the model
    model_ann_kfold = ANN_Creation(x_train=x_train,␣
↪y_train_cat=to_categorical(y_train, num_classes=10),
                                    no_hidden_layer=no_hiddenlayer,␣
↪dropout_rate=dropout_rate)

    model_ann_kfold.compile(optimizer= optimizers,
                    loss=loss_val,
                    metrics=[metrics_val])


    k_folds = k_fold
    # Initialize the StratifiedKFold object
    skf = StratifiedKFold(n_splits=k_folds, shuffle=True, random_state=42)


    total_time = 0
    losses = []
    accuracies = []
    val_losses = []
    val_accuracies = []

    # Iterate over the folds
    for fold, (trn_index, val_index) in enumerate(skf.split(X=x_train,␣
↪y=y_train)):
#         print(f'Fold {fold+1}')

        # Split the data into training and validation sets
        x_trn, x_val = x_train[trn_index], x_train[val_index]
        y_trn, y_val = y_train[trn_index], y_train[val_index]

        # Convert y-labels to one-hot encoded vectors
        y_trn = to_categorical(y_trn, num_classes=10)
        y_val = to_categorical(y_val, num_classes=10)


        # Train the model on this fold and measure the time taken
        start_time = time.time()

        early_stop = EarlyStopping(monitor='val_loss', patience=3)

        model_ann_kfold.fit(x=x_trn,
                    y=y_trn,
                    epochs=20,
                    batch_size=32,
                    validation_data=(x_val, y_val),
```

```
                  verbose=0,
                  callbacks=[early_stop]
                 )

        if display_time == 'X':
            print(f'Fold {fold+1}')
            end_time = time.time()
            fold_time = end_time - start_time
            total_time += fold_time
            print(f"Time taken for this fold: {fold_time} seconds")

        # Collect loss and accuracy measures
        fold_loss = model_ann_kfold.history.history['loss']
        fold_accuracy = model_ann_kfold.history.history['accuracy']
        fold_val_loss = model_ann_kfold.history.history['val_loss']
        fold_val_accuracy = model_ann_kfold.history.history['val_accuracy']
        losses.append(fold_loss)
        accuracies.append(fold_accuracy)
        val_losses.append(fold_val_loss)
        val_accuracies.append(fold_val_accuracy)


    loss_cv = average_loss_accuracy(losses, accuracies, val_losses,␣
 ↪val_accuracies)

    if display_time == 'X':
        print(f"Total time taken for training across all folds: {total_time}␣
 ↪seconds")

    return model_ann_kfold, loss_cv
```

Training ANN with SGD along Cross-Validation

[433]:
```
model_ann_sgd, loss_cv_sgd = ann_with_cross_val(x_train, y_train,␣
 ↪no_hiddenlayer=5,
                                                dropout_rate=0.0, k_fold=5,␣
 ↪optimizers ='sgd',
                                 loss_val='categorical_crossentropy',␣
 ↪metrics_val='accuracy', display_time = 'X')
```

```
Fold 1
Time taken for this fold: 83.98889684677124 seconds
Fold 2
Time taken for this fold: 46.82399106025696 seconds
Fold 3
Time taken for this fold: 16.85518479347229 seconds
Fold 4
```

```
Time taken for this fold: 21.656418085098267 seconds
Fold 5
Time taken for this fold: 28.878655910491943 seconds
Total time taken for training across all folds: 198.2031466960907 seconds
```

[434]:
```
#Testing with above model
print(model_ann_sgd.metrics_names)
print(model_ann_sgd.evaluate(x=x_test, y=y_test_cat, verbose = 0))
```

```
['loss', 'accuracy']
[0.8438830971717834, 0.8422999978065491]
```

[ ]:
```

```

Justify your choice of optimizers and regulizations used and the hyperparameters tuned

Optimizer

- Among all optimizers, ADAM, RMSprop & SGD are widely used. Out of them, ADAM is best optimizer it trains in less time and more efficiently.

Regularization

L2 Regularization can be applied to Dense layers.

At every iteration, **Dropout** randomly selects some nodes and removes them along with all of their incoming and outgoing connections.

Early Stopping is a kind of cross-validation strategy where we keep one part of the training set as the validation set. When we see that performance on validation set is getting worse, we immediately stop training on the model.

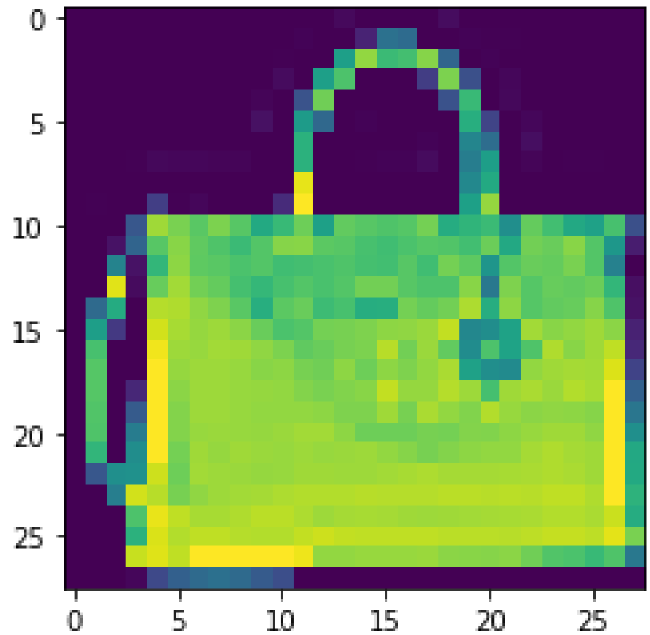
Other Hyperparameters

- loss='categorical_crossentropy' is used for calcuation of loss for multi-class categorical variable
- metrics: List of metrics('accuracy', 'mse') to be evaluated by the model during training & testing. 'accuracy' is used here.

6 6. Test the model - 0.5 marks

Prediction of Test Image[15] by ANN Model (model_ann) as defined in section 4.2

[435]:
```
test_image = x_test[30]
plt.imshow(test_image)
plt.show()
```



[436]:
```
test_image_pred = model_ann.predict(test_image.reshape(1,28,28,1))
test_image_pred = np.argmax(test_image_pred, axis=1)

print(f'Predicted Value for above Image is {test_image_pred[0]} & it is image␣
↪of {class_names[test_image_pred[0]]}')
```

```
1/1 [==============================] - 0s 265ms/step
Predicted Value for above Image is 8 & it is image of Bag
```

[ ]:
```

```

7 7. Intermediate result - Score: 1 mark

1. Plot the training and validation accuracy history.
2. Plot the training and validation loss history.
3. Report the testing accuracy and loss.
4. Show Confusion Matrix for testing dataset.
5. Report values for preformance study metrics like accuracy, precision, recall, F1 Score.

All below results are received after fitting with simple ANN Model (model_ann) as defined in section 4.2

[437]:
```
print(f'Summary of loss & Accuracy of Model \n\n {loss}')
```

```
Summary of loss & Accuracy of Model
```

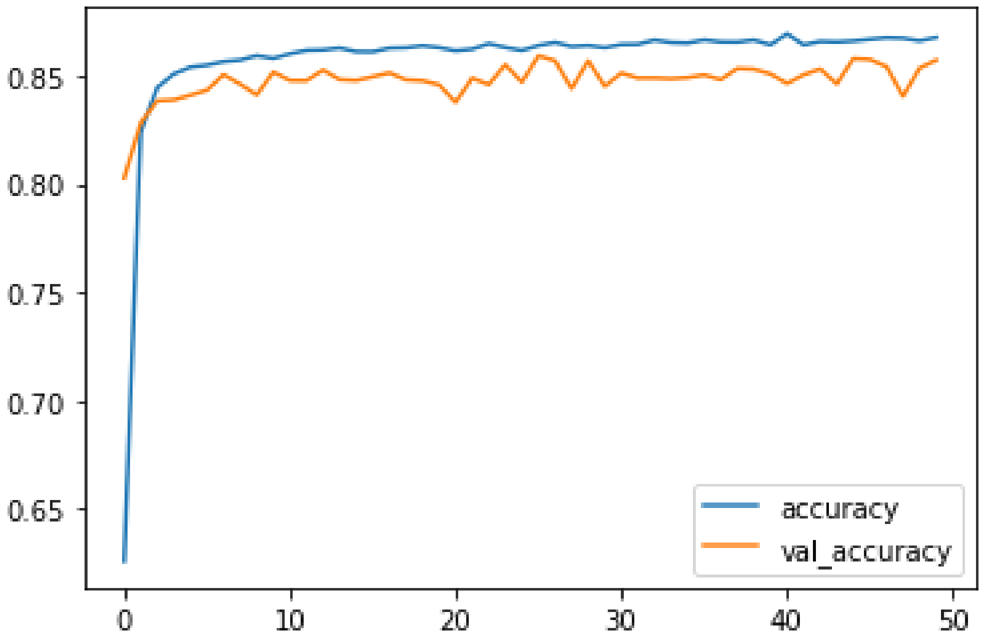
	loss	accuracy	val_loss	val_accuracy
0	2.595598	0.625767	1.280720	0.8027
1	1.157360	0.824033	1.081593	0.8282
2	1.017141	0.844550	0.999080	0.8384
3	0.948368	0.850833	0.953915	0.8388
4	0.903644	0.853950	0.918282	0.8410
5	0.875573	0.854867	0.898631	0.8433
6	0.851277	0.856517	0.859239	0.8505
7	0.832805	0.857200	0.864257	0.8461
8	0.813569	0.859250	0.844696	0.8411
9	0.805225	0.857983	0.815784	0.8517
10	0.785591	0.860100	0.819873	0.8478
11	0.773240	0.861700	0.799401	0.8477
12	0.763800	0.861900	0.797719	0.8526
13	0.752376	0.862800	0.792464	0.8482
14	0.747804	0.861100	0.776313	0.8478
15	0.741863	0.861033	0.767558	0.8495
16	0.730523	0.862750	0.769708	0.8513
17	0.726407	0.862917	0.757823	0.8481
18	0.717385	0.863833	0.753483	0.8477
19	0.711996	0.863000	0.747854	0.8458
20	0.711908	0.861517	0.768582	0.8377
21	0.703950	0.862283	0.750594	0.8489
22	0.696769	0.864767	0.743608	0.8459
23	0.694620	0.863050	0.730153	0.8551
24	0.694100	0.861667	0.735649	0.8470
25	0.683810	0.863933	0.707320	0.8592
26	0.679385	0.865333	0.706372	0.8569
27	0.680664	0.863467	0.729935	0.8440
28	0.675377	0.863917	0.700643	0.8569
29	0.674605	0.862950	0.720618	0.8451
30	0.671822	0.864400	0.712720	0.8512
31	0.664559	0.864483	0.700137	0.8489
32	0.657447	0.866467	0.702910	0.8489
33	0.658702	0.865317	0.698190	0.8486
34	0.658874	0.865067	0.702073	0.8491
35	0.652631	0.866533	0.684902	0.8503
36	0.650120	0.865600	0.701996	0.8482
37	0.650247	0.865517	0.684696	0.8533
38	0.648273	0.866483	0.681625	0.8530
39	0.647034	0.864250	0.685605	0.8508
40	0.638908	0.869333	0.701989	0.8464
41	0.642083	0.864217	0.673670	0.8504
42	0.638776	0.865817	0.667254	0.8531
43	0.638898	0.865583	0.680821	0.8462
44	0.634760	0.866033	0.659708	0.8579
45	0.633767	0.866833	0.662458	0.8575
46	0.628420	0.867550	0.659213	0.8542

```
47  0.629550  0.867317  0.695198        0.8405
48  0.630877  0.866117  0.674351        0.8535
49  0.628132  0.867667  0.670012        0.8572
```

Plot of Training & Validation Accuracy History

[438]:
```
loss[['accuracy','val_accuracy']].plot()
```

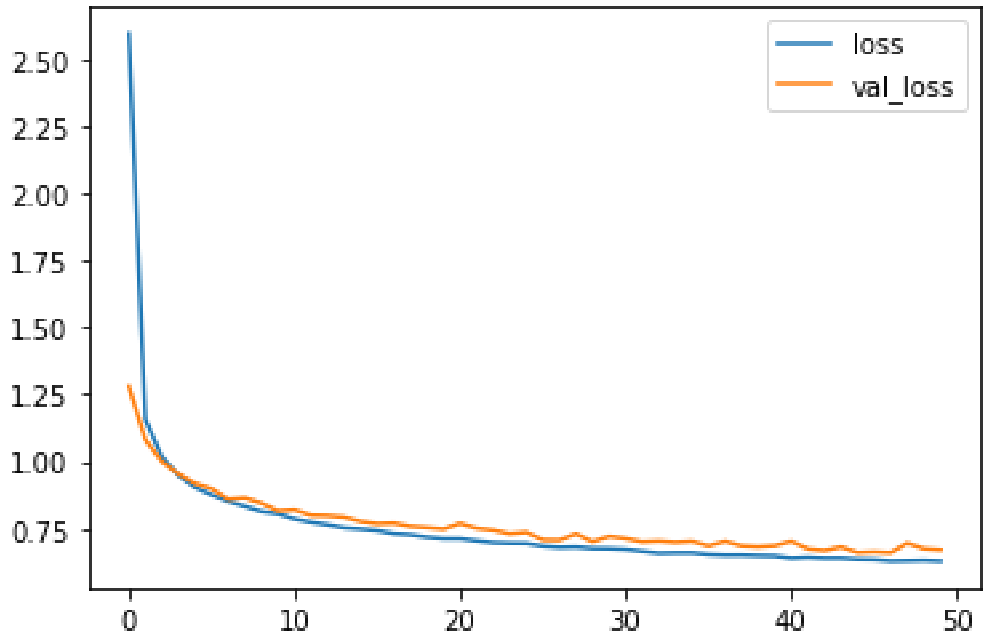
[438]: <AxesSubplot:>



Plot of Training & Validation Loss History

[439]:
```
loss[['loss','val_loss']].plot()
```

[439]: <AxesSubplot:>



Report of Testing Accuracy and Loss

[440]:
```
print(model_ann.metrics_names)
print(model_ann.evaluate(x=x_test, y=y_test_cat, verbose = 0))
```

```
['loss', 'accuracy']
[0.6700122356414795, 0.857200026512146]
```

Display of Confusion Matrix for Testing Dataset

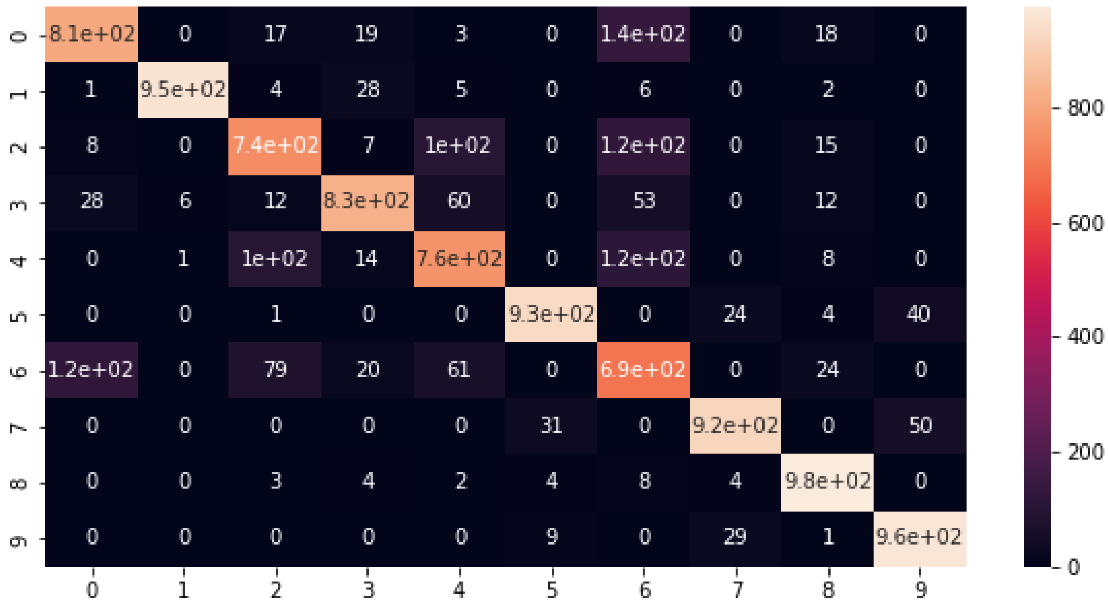
[441]:
```
y_test_predict = model_ann.predict(x=x_test, verbose=0)
y_test_predict = np.argmax(y_test_predict, axis=1)
```

[442]:
```
print(confusion_matrix(y_true=y_test, y_pred=y_test_predict))
```

```
[[806   0  17  19   3   0 137   0  18   0]
 [  1 954   4  28   5   0   6   0   2   0]
 [  8   0 745   7 102   0 123   0  15   0]
 [ 28   6  12 829  60   0  53   0  12   0]
 [  0   1 102  14 760   0 115   0   8   0]
 [  0   0   1   0   0 931   0  24   4  40]
 [124   0  79  20  61   0 692   0  24   0]
 [  0   0   0   0   0  31   0 919   0  50]
 [  0   0   3   4   2   4   8   4 975   0]
 [  0   0   0   0   0   9   0  29   1 961]]
```

[443]:

```
plt.figure(figsize=(10,5))
sns.heatmap(confusion_matrix(y_test, y_test_predict), annot=True)
plt.show()
```



Report values for preformance study metrics like accuracy, precision, recall, F1 Score

[444]:

```
print(classification_report(y_true=y_test, y_pred=y_test_predict))
```

```
              precision    recall  f1-score   support

           0       0.83      0.81      0.82      1000
           1       0.99      0.95      0.97      1000
           2       0.77      0.74      0.76      1000
           3       0.90      0.83      0.86      1000
           4       0.77      0.76      0.76      1000
           5       0.95      0.93      0.94      1000
           6       0.61      0.69      0.65      1000
           7       0.94      0.92      0.93      1000
           8       0.92      0.97      0.95      1000
           9       0.91      0.96      0.94      1000

    accuracy                           0.86     10000
   macro avg       0.86      0.86      0.86     10000
weighted avg       0.86      0.86      0.86     10000
```

[ ]:

8 8. Model architecture - Score: 1 mark

Modify the architecture designed in section 4.1

1. by decreasing one layer
2. by increasing one layer

For example, if the architecture in 4.1 has 5 layers, then 8.1 should have 4 layers and 8.2 should have 6 layers.

Plot the comparison of the training and validation accuracy of the three architecures (4.1, 8.1 and 8.2)

ANN Model Modification with one less Hidden Layer

Model 4.1 has total 6 layers (excluding input layer) inlcuding 5 hidden layeres. So, Model 8.1 should have total 5(=6-1) layers inlcuding 4(=5-1) hidden layeres.

[445]:
```
model_ann_minus = ANN_Creation(x_train=x_train, y_train_cat=y_train_cat,␣
↪no_hidden_layer=4, dropout_rate=0.0)
```

[446]:
```
model_ann_minus.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
```

[447]:
```
model_ann_minus.fit(x=x_train,
              y=y_train_cat,
              epochs=100,
              batch_size=200,
              validation_data=(x_test, y_test_cat),
              verbose=2,
              callbacks=[early_stop]
             )
```

```
Epoch 1/100
300/300 - 2s - loss: 2.2947 - accuracy: 0.6974 - val_loss: 1.2363 -
val_accuracy: 0.8091 - 2s/epoch - 8ms/step
Epoch 2/100
300/300 - 1s - loss: 1.1007 - accuracy: 0.8332 - val_loss: 1.0287 -
val_accuracy: 0.8356 - 854ms/epoch - 3ms/step
Epoch 3/100
300/300 - 1s - loss: 0.9774 - accuracy: 0.8435 - val_loss: 0.9702 -
val_accuracy: 0.8287 - 946ms/epoch - 3ms/step
Epoch 4/100
300/300 - 1s - loss: 0.9058 - accuracy: 0.8487 - val_loss: 0.8991 -
val_accuracy: 0.8420 - 1s/epoch - 3ms/step
Epoch 5/100
300/300 - 1s - loss: 0.8594 - accuracy: 0.8529 - val_loss: 0.9001 -
val_accuracy: 0.8293 - 1s/epoch - 3ms/step
Epoch 6/100
```

```
300/300 - 1s - loss: 0.8261 - accuracy: 0.8556 - val_loss: 0.8386 -
val_accuracy: 0.8475 - 1s/epoch - 3ms/step
Epoch 7/100
300/300 - 1s - loss: 0.7968 - accuracy: 0.8583 - val_loss: 0.8118 -
val_accuracy: 0.8472 - 1s/epoch - 3ms/step
Epoch 8/100
300/300 - 1s - loss: 0.7793 - accuracy: 0.8593 - val_loss: 0.8051 -
val_accuracy: 0.8444 - 1s/epoch - 3ms/step
Epoch 9/100
300/300 - 1s - loss: 0.7566 - accuracy: 0.8606 - val_loss: 0.7774 -
val_accuracy: 0.8516 - 1s/epoch - 3ms/step
Epoch 10/100
300/300 - 1s - loss: 0.7444 - accuracy: 0.8619 - val_loss: 0.7804 -
val_accuracy: 0.8479 - 1s/epoch - 4ms/step
Epoch 11/100
300/300 - 1s - loss: 0.7299 - accuracy: 0.8630 - val_loss: 0.7535 -
val_accuracy: 0.8480 - 1s/epoch - 5ms/step
Epoch 12/100
300/300 - 1s - loss: 0.7163 - accuracy: 0.8649 - val_loss: 0.7621 -
val_accuracy: 0.8438 - 1s/epoch - 4ms/step
Epoch 13/100
300/300 - 1s - loss: 0.7109 - accuracy: 0.8633 - val_loss: 0.7476 -
val_accuracy: 0.8459 - 1s/epoch - 4ms/step
Epoch 14/100
300/300 - 1s - loss: 0.6970 - accuracy: 0.8657 - val_loss: 0.7252 -
val_accuracy: 0.8553 - 1s/epoch - 3ms/step
Epoch 15/100
300/300 - 1s - loss: 0.6958 - accuracy: 0.8636 - val_loss: 0.7186 -
val_accuracy: 0.8537 - 1s/epoch - 3ms/step
Epoch 16/100
300/300 - 1s - loss: 0.6790 - accuracy: 0.8670 - val_loss: 0.7341 -
val_accuracy: 0.8438 - 1s/epoch - 3ms/step
Epoch 17/100
300/300 - 1s - loss: 0.6771 - accuracy: 0.8655 - val_loss: 0.6945 -
val_accuracy: 0.8568 - 1s/epoch - 4ms/step
Epoch 18/100
300/300 - 1s - loss: 0.6691 - accuracy: 0.8644 - val_loss: 0.6894 -
val_accuracy: 0.8567 - 1s/epoch - 3ms/step
Epoch 19/100
300/300 - 1s - loss: 0.6601 - accuracy: 0.8680 - val_loss: 0.7198 -
val_accuracy: 0.8472 - 1s/epoch - 4ms/step
Epoch 20/100
300/300 - 1s - loss: 0.6535 - accuracy: 0.8691 - val_loss: 0.6736 -
val_accuracy: 0.8619 - 1s/epoch - 4ms/step
Epoch 21/100
300/300 - 1s - loss: 0.6488 - accuracy: 0.8691 - val_loss: 0.6752 -
val_accuracy: 0.8589 - 1s/epoch - 3ms/step
Epoch 22/100
```

```
300/300 - 1s - loss: 0.6452 - accuracy: 0.8680 - val_loss: 0.6599 -
val_accuracy: 0.8612 - 1s/epoch - 3ms/step
Epoch 23/100
300/300 - 1s - loss: 0.6419 - accuracy: 0.8676 - val_loss: 0.6733 -
val_accuracy: 0.8567 - 1s/epoch - 3ms/step
Epoch 24/100
300/300 - 1s - loss: 0.6316 - accuracy: 0.8699 - val_loss: 0.6760 -
val_accuracy: 0.8552 - 1s/epoch - 4ms/step
Epoch 25/100
300/300 - 1s - loss: 0.6293 - accuracy: 0.8701 - val_loss: 0.6500 -
val_accuracy: 0.8590 - 1s/epoch - 5ms/step
Epoch 26/100
300/300 - 1s - loss: 0.6253 - accuracy: 0.8684 - val_loss: 0.6517 -
val_accuracy: 0.8597 - 1s/epoch - 4ms/step
Epoch 27/100
300/300 - 1s - loss: 0.6242 - accuracy: 0.8696 - val_loss: 0.6528 -
val_accuracy: 0.8568 - 1s/epoch - 4ms/step
Epoch 28/100
300/300 - 1s - loss: 0.6209 - accuracy: 0.8691 - val_loss: 0.6545 -
val_accuracy: 0.8560 - 1s/epoch - 3ms/step
```

[447]: <keras.callbacks.History at 0x1c19e07bc10>

[448]:
```python
loss_minus = pd.DataFrame(model_ann_minus.history.history)
print(loss_minus)
# loss_minus[['accuracy','val_accuracy']].plot()
# loss_minus[['loss','val_loss']].plot()
# plt.show()
```

```
        loss  accuracy  val_loss  val_accuracy
0   2.294749  0.697367  1.236348        0.8091
1   1.100708  0.833183  1.028746        0.8356
2   0.977438  0.843533  0.970240        0.8287
3   0.905776  0.848683  0.899129        0.8420
4   0.859403  0.852883  0.900093        0.8293
5   0.826097  0.855583  0.838574        0.8475
6   0.796764  0.858317  0.811797        0.8472
7   0.779277  0.859317  0.805063        0.8444
8   0.756636  0.860633  0.777396        0.8516
9   0.744386  0.861900  0.780435        0.8479
10  0.729945  0.863000  0.753506        0.8480
11  0.716266  0.864900  0.762146        0.8438
12  0.710863  0.863250  0.747602        0.8459
13  0.697034  0.865700  0.725227        0.8553
14  0.695778  0.863550  0.718641        0.8537
15  0.678977  0.866967  0.734103        0.8438
16  0.677103  0.865517  0.694499        0.8568
17  0.669149  0.864400  0.689429        0.8567
```

```
18  0.660075  0.868017  0.719824        0.8472
19  0.653507  0.869133  0.673588        0.8619
20  0.648771  0.869083  0.675155        0.8589
21  0.645236  0.868000  0.659871        0.8612
22  0.641894  0.867617  0.673280        0.8567
23  0.631595  0.869883  0.676038        0.8552
24  0.629289  0.870100  0.650025        0.8590
25  0.625332  0.868383  0.651746        0.8597
26  0.624239  0.869600  0.652834        0.8568
27  0.620857  0.869133  0.654520        0.8560
```

ANN Model Modification with one extra Hidden Layer

Model 4.1 has total 6 layers (excluding input layer) inlcuding 5 hidden layeres. So, Model 8.2 should have total 7(=6+1) layers inlcuding 6(=5+1) hidden layeres.

[449]:
```
model_ann_plus= ANN_Creation(x_train=x_train, y_train_cat=y_train_cat,␣
 ↪no_hidden_layer=6, dropout_rate=0.0)
```

[450]:
```
model_ann_plus.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
```

[451]:
```
model_ann_plus.fit(x=x_train,
              y=y_train_cat,
              epochs=100,
              batch_size=200,
              validation_data=(x_test, y_test_cat),
              verbose=2,
              callbacks=[early_stop]
             )
```

```
Epoch 1/100
300/300 - 5s - loss: 3.1948 - accuracy: 0.2986 - val_loss: 1.7808 -
val_accuracy: 0.3728 - 5s/epoch - 16ms/step
Epoch 2/100
300/300 - 1s - loss: 1.7004 - accuracy: 0.4190 - val_loss: 1.6462 -
val_accuracy: 0.4481 - 1s/epoch - 4ms/step
Epoch 3/100
300/300 - 1s - loss: 1.6166 - accuracy: 0.4452 - val_loss: 1.6007 -
val_accuracy: 0.4458 - 1s/epoch - 4ms/step
Epoch 4/100
300/300 - 1s - loss: 1.5666 - accuracy: 0.4626 - val_loss: 1.5520 -
val_accuracy: 0.5089 - 1s/epoch - 4ms/step
Epoch 5/100
300/300 - 1s - loss: 1.5253 - accuracy: 0.5110 - val_loss: 1.4867 -
val_accuracy: 0.5328 - 1s/epoch - 5ms/step
Epoch 6/100
300/300 - 2s - loss: 1.3846 - accuracy: 0.5988 - val_loss: 1.3271 -
```

```
val_accuracy: 0.6371 - 2s/epoch - 6ms/step
Epoch 7/100
300/300 - 2s - loss: 1.2834 - accuracy: 0.6478 - val_loss: 1.2721 -
val_accuracy: 0.6756 - 2s/epoch - 7ms/step
Epoch 8/100
300/300 - 2s - loss: 1.2222 - accuracy: 0.6645 - val_loss: 1.2105 -
val_accuracy: 0.6732 - 2s/epoch - 6ms/step
Epoch 9/100
300/300 - 1s - loss: 1.1827 - accuracy: 0.6707 - val_loss: 1.1867 -
val_accuracy: 0.6393 - 1s/epoch - 5ms/step
Epoch 10/100
300/300 - 2s - loss: 1.1511 - accuracy: 0.6729 - val_loss: 1.1436 -
val_accuracy: 0.6871 - 2s/epoch - 5ms/step
Epoch 11/100
300/300 - 2s - loss: 1.1177 - accuracy: 0.6826 - val_loss: 1.1302 -
val_accuracy: 0.6710 - 2s/epoch - 5ms/step
Epoch 12/100
300/300 - 1s - loss: 1.1001 - accuracy: 0.6814 - val_loss: 1.1016 -
val_accuracy: 0.6858 - 1s/epoch - 5ms/step
Epoch 13/100
300/300 - 1s - loss: 1.0855 - accuracy: 0.6848 - val_loss: 1.1035 -
val_accuracy: 0.6804 - 1s/epoch - 5ms/step
Epoch 14/100
300/300 - 1s - loss: 1.0687 - accuracy: 0.6867 - val_loss: 1.0784 -
val_accuracy: 0.6928 - 1s/epoch - 5ms/step
Epoch 15/100
300/300 - 1s - loss: 1.0606 - accuracy: 0.6905 - val_loss: 1.0778 -
val_accuracy: 0.6828 - 1s/epoch - 5ms/step
Epoch 16/100
300/300 - 2s - loss: 1.0545 - accuracy: 0.6893 - val_loss: 1.0822 -
val_accuracy: 0.6742 - 2s/epoch - 6ms/step
Epoch 17/100
300/300 - 2s - loss: 1.0392 - accuracy: 0.6939 - val_loss: 1.0545 -
val_accuracy: 0.6906 - 2s/epoch - 6ms/step
Epoch 18/100
300/300 - 2s - loss: 1.0306 - accuracy: 0.6956 - val_loss: 1.0536 -
val_accuracy: 0.6905 - 2s/epoch - 5ms/step
Epoch 19/100
300/300 - 1s - loss: 1.0256 - accuracy: 0.6998 - val_loss: 1.0426 -
val_accuracy: 0.7026 - 1s/epoch - 5ms/step
Epoch 20/100
300/300 - 1s - loss: 1.0167 - accuracy: 0.7022 - val_loss: 1.0384 -
val_accuracy: 0.6993 - 1s/epoch - 5ms/step
Epoch 21/100
300/300 - 1s - loss: 1.0138 - accuracy: 0.7017 - val_loss: 1.0361 -
val_accuracy: 0.7046 - 1s/epoch - 5ms/step
Epoch 22/100
300/300 - 1s - loss: 1.0077 - accuracy: 0.7012 - val_loss: 1.0404 -
```

```
val_accuracy: 0.6687 - 1s/epoch - 5ms/step
Epoch 23/100
300/300 - 1s - loss: 0.9976 - accuracy: 0.7057 - val_loss: 1.0258 -
val_accuracy: 0.7011 - 1s/epoch - 5ms/step
Epoch 24/100
300/300 - 1s - loss: 0.9954 - accuracy: 0.7073 - val_loss: 1.0159 -
val_accuracy: 0.7006 - 1s/epoch - 5ms/step
Epoch 25/100
300/300 - 1s - loss: 0.9897 - accuracy: 0.7066 - val_loss: 1.0070 -
val_accuracy: 0.7019 - 1s/epoch - 5ms/step
Epoch 26/100
300/300 - 2s - loss: 0.9812 - accuracy: 0.7085 - val_loss: 1.0082 -
val_accuracy: 0.7064 - 2s/epoch - 6ms/step
Epoch 27/100
300/300 - 2s - loss: 0.9781 - accuracy: 0.7079 - val_loss: 1.0045 -
val_accuracy: 0.7023 - 2s/epoch - 6ms/step
Epoch 28/100
300/300 - 2s - loss: 0.9730 - accuracy: 0.7114 - val_loss: 1.0069 -
val_accuracy: 0.7006 - 2s/epoch - 6ms/step
Epoch 29/100
300/300 - 1s - loss: 0.9725 - accuracy: 0.7100 - val_loss: 0.9966 -
val_accuracy: 0.7065 - 1s/epoch - 5ms/step
Epoch 30/100
300/300 - 1s - loss: 0.9626 - accuracy: 0.7132 - val_loss: 0.9989 -
val_accuracy: 0.7004 - 1s/epoch - 5ms/step
Epoch 31/100
300/300 - 1s - loss: 0.9465 - accuracy: 0.7193 - val_loss: 0.9790 -
val_accuracy: 0.7095 - 1s/epoch - 5ms/step
Epoch 32/100
300/300 - 1s - loss: 0.9217 - accuracy: 0.7308 - val_loss: 0.9240 -
val_accuracy: 0.7351 - 1s/epoch - 5ms/step
Epoch 33/100
300/300 - 1s - loss: 0.8918 - accuracy: 0.7481 - val_loss: 0.9295 -
val_accuracy: 0.7254 - 1s/epoch - 5ms/step
Epoch 34/100
300/300 - 1s - loss: 0.8854 - accuracy: 0.7553 - val_loss: 0.9057 -
val_accuracy: 0.7512 - 1s/epoch - 5ms/step
Epoch 35/100
300/300 - 1s - loss: 0.8645 - accuracy: 0.7709 - val_loss: 0.8796 -
val_accuracy: 0.7806 - 1s/epoch - 5ms/step
Epoch 36/100
300/300 - 2s - loss: 0.8491 - accuracy: 0.7904 - val_loss: 0.8689 -
val_accuracy: 0.7874 - 2s/epoch - 6ms/step
Epoch 37/100
300/300 - 2s - loss: 0.8434 - accuracy: 0.7948 - val_loss: 0.8968 -
val_accuracy: 0.7817 - 2s/epoch - 6ms/step
Epoch 38/100
300/300 - 2s - loss: 0.8376 - accuracy: 0.7982 - val_loss: 0.8818 -
```

```
val_accuracy: 0.7870 - 2s/epoch - 6ms/step
Epoch 39/100
300/300 - 1s - loss: 0.8244 - accuracy: 0.8005 - val_loss: 0.9038 -
val_accuracy: 0.7753 - 1s/epoch - 5ms/step
```

[451]:
```
<keras.callbacks.History at 0x1c19e61cbb0>
```

[452]:
```python
loss_plus = pd.DataFrame(model_ann_plus.history.history)
print(loss_plus)
# loss_plus[['accuracy','val_accuracy']].plot()
# loss_plus[['loss','val_loss']].plot()
# plt.show()
```

```
        loss  accuracy  val_loss  val_accuracy
0   3.194811  0.298600  1.780775        0.3728
1   1.700378  0.418983  1.646151        0.4481
2   1.616620  0.445200  1.600713        0.4458
3   1.566601  0.462583  1.551984        0.5089
4   1.525334  0.511000  1.486699        0.5328
5   1.384595  0.598783  1.327074        0.6371
6   1.283423  0.647750  1.272098        0.6756
7   1.222219  0.664517  1.210536        0.6732
8   1.182714  0.670683  1.186711        0.6393
9   1.151095  0.672867  1.143593        0.6871
10  1.117697  0.682567  1.130242        0.6710
11  1.100091  0.681400  1.101587        0.6858
12  1.085480  0.684850  1.103485        0.6804
13  1.068747  0.686683  1.078386        0.6928
14  1.060593  0.690467  1.077799        0.6828
15  1.054464  0.689283  1.082196        0.6742
16  1.039207  0.693900  1.054511        0.6906
17  1.030580  0.695583  1.053554        0.6905
18  1.025628  0.699767  1.042603        0.7026
19  1.016686  0.702233  1.038365        0.6993
20  1.013814  0.701733  1.036108        0.7046
21  1.007728  0.701217  1.040376        0.6687
22  0.997550  0.705733  1.025768        0.7011
23  0.995356  0.707317  1.015949        0.7006
24  0.989712  0.706633  1.007027        0.7019
25  0.981199  0.708517  1.008227        0.7064
26  0.978127  0.707950  1.004532        0.7023
27  0.973014  0.711417  1.006912        0.7006
28  0.972504  0.710033  0.996620        0.7065
29  0.962619  0.713183  0.998855        0.7004
30  0.946547  0.719267  0.978951        0.7095
31  0.921700  0.730850  0.923995        0.7351
32  0.891800  0.748150  0.929490        0.7254
33  0.885416  0.755350  0.905651        0.7512
```

34	0.864547	0.770883	0.879635	0.7806
35	0.849116	0.790400	0.868892	0.7874
36	0.843419	0.794833	0.896772	0.7817
37	0.837616	0.798150	0.881816	0.7870
38	0.824403	0.800467	0.903807	0.7753

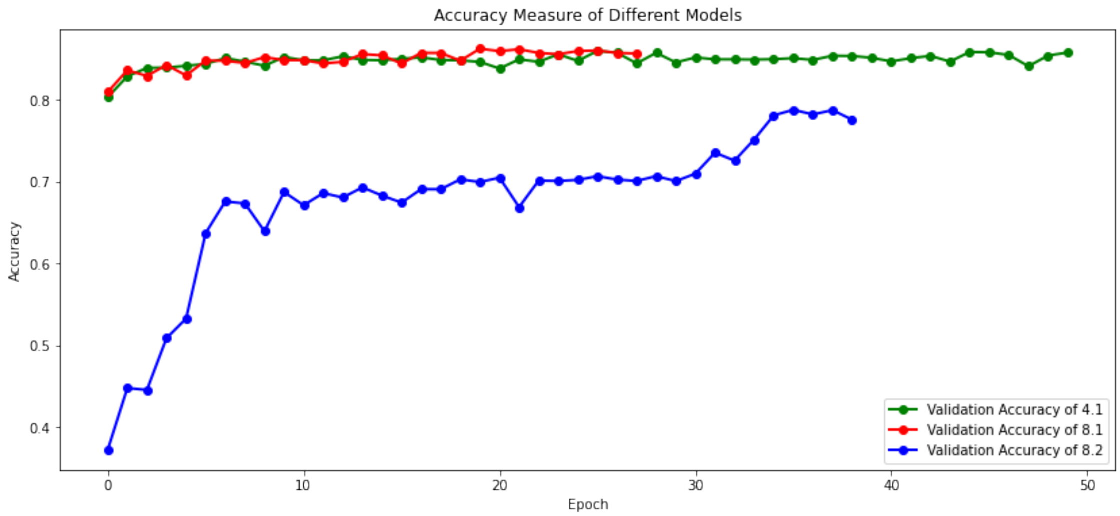
Plot Comparison of Training and Validation Accuracy of 3 architecures (4.1, 8.1, 8.2)

[453]:
```
loss['index']=loss.index
loss_minus['index']=loss_minus.index
loss_plus['index']=loss_plus.index
```

[454]:
```
plt.figure(figsize = (14, 6))
plt.plot(loss.iloc[:,4],loss.iloc[:,3], 'go-', label='Validation Accuracy of 4.
 ↪1', linewidth=2)
plt.plot(loss_minus.iloc[:,4],loss_minus.iloc[:,3], 'ro-', label='Validation␣
 ↪Accuracy of 8.1', linewidth=2)
plt.plot(loss_plus.iloc[:,4],loss_plus.iloc[:,3], 'bo-', label='Validation␣
 ↪Accuracy of 8.2', linewidth=2)

plt.title('Accuracy Measure of Different Models')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



[455]:
```
am_max_4_1 = max(loss.iloc[:,1])
am_max_8_1 = max(loss_minus.iloc[:,1])
am_max_8_2 = max(loss_plus.iloc[:,1])
```

```
am_max = max(am_max_4_1,am_max_8_1, am_max_8_2)

if  am_max == am_max_4_1:
    print(f'Model 4.1 is having highest Validation Accuracy {am_max_4_1}')
elif am_max == am_max_8_1:
    print(f'Model 8.1 is having highest Validation Accuracy {am_max_8_1}')
else:
    print(f'Model 8.2 is having highest Validation Accuracy {am_max_8_2}')
```

```
Model 8.1 is having highest Validation Accuracy 0.8701000213623047
```

[456]:
```
# am_max_4_1, am_max_8_1, am_max_8_2
```

9 9. Regularisations - Score: 1 mark

Modify the architecture designed in section 4.1

1. Dropout of ratio 0.25
2. Dropout of ratio 0.25 with L2 regulariser with factor 1e−04.

Plot the comparison of the training and validation accuracy of the three (4.1, 9.1 and 9.2)

ANN Model Modification with Dropout Rate

[457]:
```
model_ann_do = ANN_Creation(x_train=x_train, y_train_cat=y_train_cat,␣
 ↪no_hidden_layer=5, dropout_rate=0.25)
```

[458]:
```
model_ann_do.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
```

[459]:
```
model_ann_do.fit(x=x_train,
              y=y_train_cat,
              epochs=100,
              batch_size=200,
              validation_data=(x_test, y_test_cat),
              verbose=2,
              callbacks=[early_stop]
             )
```

```
Epoch 1/100
300/300 - 3s - loss: 3.1540 - accuracy: 0.1985 - val_loss: 1.8931 -
val_accuracy: 0.3117 - 3s/epoch - 11ms/step
Epoch 2/100
300/300 - 2s - loss: 1.8435 - accuracy: 0.3112 - val_loss: 1.7270 -
val_accuracy: 0.3935 - 2s/epoch - 5ms/step
Epoch 3/100
300/300 - 2s - loss: 1.7526 - accuracy: 0.3531 - val_loss: 1.6517 -
```

```
val_accuracy: 0.4294 - 2s/epoch - 6ms/step
Epoch 4/100
300/300 - 2s - loss: 1.7056 - accuracy: 0.3687 - val_loss: 1.6323 -
val_accuracy: 0.4277 - 2s/epoch - 7ms/step
Epoch 5/100
300/300 - 2s - loss: 1.6742 - accuracy: 0.3828 - val_loss: 1.5889 -
val_accuracy: 0.4131 - 2s/epoch - 8ms/step
Epoch 6/100
300/300 - 2s - loss: 1.6479 - accuracy: 0.3910 - val_loss: 1.5709 -
val_accuracy: 0.4458 - 2s/epoch - 8ms/step
Epoch 7/100
300/300 - 2s - loss: 1.6078 - accuracy: 0.4340 - val_loss: 1.4624 -
val_accuracy: 0.4950 - 2s/epoch - 7ms/step
Epoch 8/100
300/300 - 2s - loss: 1.5269 - accuracy: 0.4705 - val_loss: 1.3800 -
val_accuracy: 0.5301 - 2s/epoch - 7ms/step
Epoch 9/100
300/300 - 2s - loss: 1.4253 - accuracy: 0.5102 - val_loss: 1.2639 -
val_accuracy: 0.6164 - 2s/epoch - 7ms/step
Epoch 10/100
300/300 - 2s - loss: 1.3511 - accuracy: 0.5540 - val_loss: 1.2233 -
val_accuracy: 0.6205 - 2s/epoch - 7ms/step
Epoch 11/100
300/300 - 2s - loss: 1.2977 - accuracy: 0.5818 - val_loss: 1.1652 -
val_accuracy: 0.6448 - 2s/epoch - 6ms/step
Epoch 12/100
300/300 - 2s - loss: 1.2531 - accuracy: 0.6211 - val_loss: 1.1145 -
val_accuracy: 0.7005 - 2s/epoch - 6ms/step
Epoch 13/100
300/300 - 2s - loss: 1.2145 - accuracy: 0.6518 - val_loss: 1.0711 -
val_accuracy: 0.7304 - 2s/epoch - 8ms/step
Epoch 14/100
300/300 - 2s - loss: 1.1801 - accuracy: 0.6705 - val_loss: 1.0348 -
val_accuracy: 0.7190 - 2s/epoch - 8ms/step
Epoch 15/100
300/300 - 2s - loss: 1.1588 - accuracy: 0.6763 - val_loss: 1.0081 -
val_accuracy: 0.7308 - 2s/epoch - 6ms/step
Epoch 16/100
300/300 - 2s - loss: 1.1430 - accuracy: 0.6794 - val_loss: 0.9962 -
val_accuracy: 0.7357 - 2s/epoch - 6ms/step
Epoch 17/100
300/300 - 2s - loss: 1.1346 - accuracy: 0.6834 - val_loss: 0.9995 -
val_accuracy: 0.7301 - 2s/epoch - 6ms/step
Epoch 18/100
300/300 - 2s - loss: 1.1202 - accuracy: 0.6848 - val_loss: 0.9732 -
val_accuracy: 0.7314 - 2s/epoch - 6ms/step
Epoch 19/100
300/300 - 2s - loss: 1.1099 - accuracy: 0.6853 - val_loss: 0.9821 -
```

```
val_accuracy: 0.7340 - 2s/epoch - 7ms/step
Epoch 20/100
300/300 - 2s - loss: 1.1073 - accuracy: 0.6879 - val_loss: 0.9647 -
val_accuracy: 0.7392 - 2s/epoch - 7ms/step
Epoch 21/100
300/300 - 2s - loss: 1.0956 - accuracy: 0.6935 - val_loss: 0.9584 -
val_accuracy: 0.7310 - 2s/epoch - 8ms/step
Epoch 22/100
300/300 - 2s - loss: 1.0922 - accuracy: 0.6947 - val_loss: 0.9703 -
val_accuracy: 0.7280 - 2s/epoch - 8ms/step
Epoch 23/100
300/300 - 2s - loss: 1.0823 - accuracy: 0.7006 - val_loss: 0.9350 -
val_accuracy: 0.7639 - 2s/epoch - 6ms/step
Epoch 24/100
300/300 - 2s - loss: 1.0766 - accuracy: 0.6995 - val_loss: 0.9444 -
val_accuracy: 0.7493 - 2s/epoch - 6ms/step
Epoch 25/100
300/300 - 2s - loss: 1.0768 - accuracy: 0.7001 - val_loss: 0.9472 -
val_accuracy: 0.7366 - 2s/epoch - 6ms/step
Epoch 26/100
300/300 - 2s - loss: 1.0747 - accuracy: 0.7036 - val_loss: 0.9379 -
val_accuracy: 0.7534 - 2s/epoch - 6ms/step
```

[459]: <keras.callbacks.History at 0x1c1226d3bb0>

[460]:
```
loss_do = pd.DataFrame(model_ann_do.history.history)
print(loss_do)
```

```
        loss  accuracy  val_loss  val_accuracy
0   3.153980  0.198483  1.893097        0.3117
1   1.843500  0.311217  1.727014        0.3935
2   1.752647  0.353083  1.651686        0.4294
3   1.705647  0.368683  1.632329        0.4277
4   1.674248  0.382750  1.588896        0.4131
5   1.647946  0.391017  1.570868        0.4458
6   1.607790  0.433983  1.462435        0.4950
7   1.526917  0.470517  1.380020        0.5301
8   1.425301  0.510167  1.263890        0.6164
9   1.351088  0.554000  1.223272        0.6205
10  1.297695  0.581767  1.165217        0.6448
11  1.253146  0.621050  1.114478        0.7005
12  1.214524  0.651767  1.071061        0.7304
13  1.180147  0.670517  1.034831        0.7190
14  1.158828  0.676300  1.008065        0.7308
15  1.143036  0.679450  0.996242        0.7357
16  1.134577  0.683450  0.999490        0.7301
17  1.120230  0.684750  0.973155        0.7314
18  1.109926  0.685267  0.982101        0.7340
```

19	1.107346	0.687900	0.964736	0.7392
20	1.095647	0.693517	0.958399	0.7310
21	1.092220	0.694733	0.970308	0.7280
22	1.082271	0.700583	0.935006	0.7639
23	1.076632	0.699500	0.944431	0.7493
24	1.076813	0.700133	0.947171	0.7366
25	1.074659	0.703567	0.937907	0.7534

ANN Model Modification with Dropout Rate & L2 Regulariser

Dropout of ratio 0.25 with L2 regulariser with factor 1e−04.

[461]:
```
model_ann_dor = ANN_Creation(x_train=x_train, y_train_cat=y_train_cat,
                             no_hidden_layer=5, dropout_rate=0.25,␣
 ↪kernel_regularizer= 0.0001)
```

[462]:
```
model_ann_dor.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
```

[463]:
```
model_ann_dor.fit(x=x_train,
              y=y_train_cat,
              epochs=100,
              batch_size=200,
              validation_data=(x_test, y_test_cat),
              verbose=2,
              callbacks=[early_stop]
             )
```

```
Epoch 1/100
300/300 - 3s - loss: 1.7100 - accuracy: 0.4415 - val_loss: 1.0889 -
val_accuracy: 0.6347 - 3s/epoch - 11ms/step
Epoch 2/100
300/300 - 2s - loss: 1.1532 - accuracy: 0.6162 - val_loss: 0.9618 -
val_accuracy: 0.6531 - 2s/epoch - 7ms/step
Epoch 3/100
300/300 - 2s - loss: 1.0648 - accuracy: 0.6291 - val_loss: 0.9205 -
val_accuracy: 0.6568 - 2s/epoch - 7ms/step
Epoch 4/100
300/300 - 2s - loss: 1.0208 - accuracy: 0.6343 - val_loss: 0.9035 -
val_accuracy: 0.6589 - 2s/epoch - 6ms/step
Epoch 5/100
300/300 - 2s - loss: 0.9987 - accuracy: 0.6367 - val_loss: 0.8928 -
val_accuracy: 0.6566 - 2s/epoch - 6ms/step
Epoch 6/100
300/300 - 2s - loss: 0.9780 - accuracy: 0.6410 - val_loss: 0.8728 -
val_accuracy: 0.6624 - 2s/epoch - 6ms/step
Epoch 7/100
300/300 - 2s - loss: 0.9618 - accuracy: 0.6432 - val_loss: 0.8703 -
```

```
val_accuracy: 0.6637 - 2s/epoch - 6ms/step
Epoch 8/100
300/300 - 2s - loss: 0.9474 - accuracy: 0.6462 - val_loss: 0.8571 -
val_accuracy: 0.6616 - 2s/epoch - 6ms/step
Epoch 9/100
300/300 - 2s - loss: 0.9399 - accuracy: 0.6471 - val_loss: 0.8545 -
val_accuracy: 0.6624 - 2s/epoch - 7ms/step
Epoch 10/100
300/300 - 2s - loss: 0.9332 - accuracy: 0.6477 - val_loss: 0.8431 -
val_accuracy: 0.6670 - 2s/epoch - 8ms/step
Epoch 11/100
300/300 - 2s - loss: 0.9293 - accuracy: 0.6496 - val_loss: 0.8502 -
val_accuracy: 0.6630 - 2s/epoch - 8ms/step
Epoch 12/100
300/300 - 2s - loss: 0.9217 - accuracy: 0.6478 - val_loss: 0.8343 -
val_accuracy: 0.6711 - 2s/epoch - 6ms/step
Epoch 13/100
300/300 - 2s - loss: 0.9139 - accuracy: 0.6529 - val_loss: 0.8291 -
val_accuracy: 0.6674 - 2s/epoch - 6ms/step
Epoch 14/100
300/300 - 2s - loss: 0.9076 - accuracy: 0.6566 - val_loss: 0.8353 -
val_accuracy: 0.6620 - 2s/epoch - 6ms/step
Epoch 15/100
300/300 - 2s - loss: 0.9059 - accuracy: 0.6615 - val_loss: 0.8108 -
val_accuracy: 0.7199 - 2s/epoch - 6ms/step
Epoch 16/100
300/300 - 2s - loss: 0.8769 - accuracy: 0.6995 - val_loss: 0.7406 -
val_accuracy: 0.7399 - 2s/epoch - 6ms/step
Epoch 17/100
300/300 - 2s - loss: 0.8462 - accuracy: 0.7102 - val_loss: 0.7169 -
val_accuracy: 0.7459 - 2s/epoch - 7ms/step
Epoch 18/100
300/300 - 2s - loss: 0.8341 - accuracy: 0.7119 - val_loss: 0.7366 -
val_accuracy: 0.7360 - 2s/epoch - 8ms/step
Epoch 19/100
300/300 - 2s - loss: 0.8176 - accuracy: 0.7177 - val_loss: 0.6866 -
val_accuracy: 0.7462 - 2s/epoch - 8ms/step
Epoch 20/100
300/300 - 2s - loss: 0.8070 - accuracy: 0.7193 - val_loss: 0.6831 -
val_accuracy: 0.7478 - 2s/epoch - 6ms/step
Epoch 21/100
300/300 - 2s - loss: 0.7946 - accuracy: 0.7228 - val_loss: 0.6764 -
val_accuracy: 0.7477 - 2s/epoch - 6ms/step
Epoch 22/100
300/300 - 2s - loss: 0.7827 - accuracy: 0.7270 - val_loss: 0.6816 -
val_accuracy: 0.7531 - 2s/epoch - 6ms/step
Epoch 23/100
300/300 - 2s - loss: 0.7758 - accuracy: 0.7271 - val_loss: 0.6628 -
```

```
val_accuracy: 0.7551 - 2s/epoch - 6ms/step
Epoch 24/100
300/300 - 2s - loss: 0.7700 - accuracy: 0.7304 - val_loss: 0.6636 -
val_accuracy: 0.7519 - 2s/epoch - 6ms/step
Epoch 25/100
300/300 - 2s - loss: 0.7689 - accuracy: 0.7287 - val_loss: 0.6701 -
val_accuracy: 0.7511 - 2s/epoch - 6ms/step
Epoch 26/100
300/300 - 2s - loss: 0.7733 - accuracy: 0.7260 - val_loss: 0.6664 -
val_accuracy: 0.7534 - 2s/epoch - 8ms/step
```

[463]:
```
<keras.callbacks.History at 0x1c19e2b8ee0>
```

[464]:
```python
loss_dor = pd.DataFrame(model_ann_dor.history.history)
print(loss_dor)
```

```
        loss  accuracy  val_loss  val_accuracy
0   1.709980  0.441500  1.088859        0.6347
1   1.153175  0.616150  0.961756        0.6531
2   1.064832  0.629067  0.920473        0.6568
3   1.020788  0.634317  0.903524        0.6589
4   0.998687  0.636667  0.892811        0.6566
5   0.978002  0.641033  0.872830        0.6624
6   0.961788  0.643217  0.870315        0.6637
7   0.947408  0.646183  0.857073        0.6616
8   0.939871  0.647117  0.854451        0.6624
9   0.933227  0.647717  0.843120        0.6670
10  0.929328  0.649583  0.850211        0.6630
11  0.921685  0.647750  0.834306        0.6711
12  0.913950  0.652917  0.829062        0.6674
13  0.907568  0.656617  0.835345        0.6620
14  0.905881  0.661450  0.810827        0.7199
15  0.876917  0.699467  0.740584        0.7399
16  0.846160  0.710217  0.716942        0.7459
17  0.834138  0.711883  0.736649        0.7360
18  0.817564  0.717700  0.686626        0.7462
19  0.806988  0.719317  0.683068        0.7478
20  0.794638  0.722750  0.676443        0.7477
21  0.782657  0.726967  0.681611        0.7531
22  0.775825  0.727067  0.662805        0.7551
23  0.770027  0.730417  0.663580        0.7519
24  0.768872  0.728750  0.670099        0.7511
25  0.773285  0.726000  0.666427        0.7534
```

Plot Comparison of Training & Validation Accuracy of 3 Models (4.1, 9.1 and 9.2)

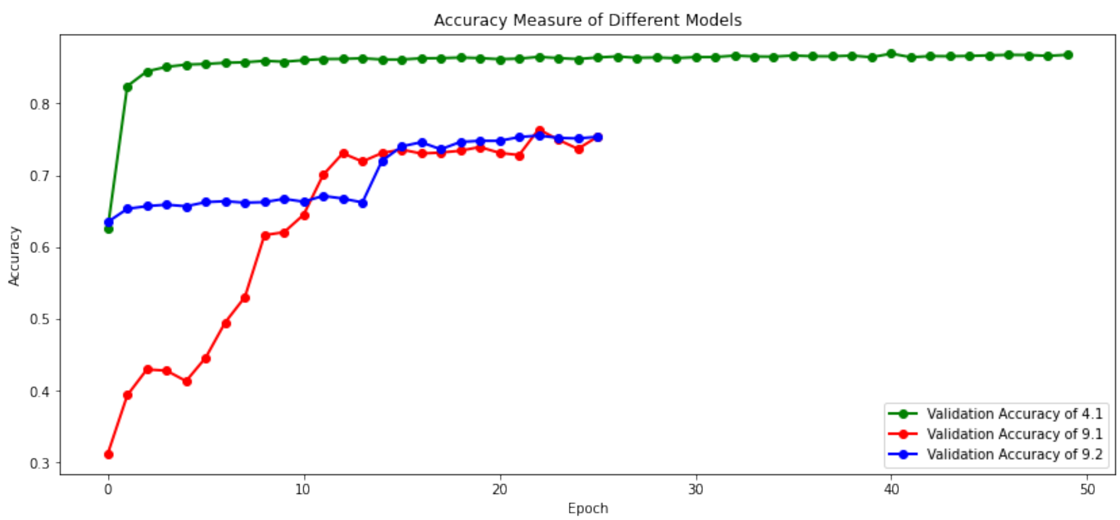
[465]:
```python
# loss['index']=loss.index
loss_do['index']=loss_do.index
```

```
loss_dor['index']=loss_dor.index
```

[466]:
```
plt.figure(figsize = (14, 6))
plt.plot(loss.iloc[:,4],loss.iloc[:,1], 'go-', label='Validation Accuracy of 4.
 ↪1', linewidth=2)
plt.plot(loss_do.iloc[:,4],loss_do.iloc[:,3], 'ro-', label='Validation Accuracy␣
 ↪of 9.1', linewidth=2)
plt.plot(loss_dor.iloc[:,4],loss_dor.iloc[:,3], 'bo-', label='Validation␣
 ↪Accuracy of 9.2', linewidth=2)

plt.title('Accuracy Measure of Different Models')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



[467]:
```
am_max_4_1 = max(loss.iloc[:,1])
am_max_9_1 = max(loss_do.iloc[:,1])
am_max_9_2 = max(loss_dor.iloc[:,1])
am_max = max(am_max_4_1,am_max_9_1, am_max_9_2)

if  am_max == am_max_4_1:
    print(f'Model 4.1 is having highest Validation Accuracy {am_max_4_1}')
elif am_max == am_max_9_1:
    print(f'Model 9.1 is having highest Validation Accuracy {am_max_9_1}')
else:
    print(f'Model 9.2 is having highest Validation Accuracy {am_max_9_2}')
```

```
Model 4.1 is having highest Validation Accuracy 0.8693333268165588
```

[468]:
```
# am_max_4_1, am_max_9_1, am_max_9_2
```

10 10. Optimisers -Score: 1 mark

Modify the code written in section 5.2

1. RMSProp with your choice of hyper parameters
2. Adam with your choice of hyper parameters

Plot the comparison of the training and validation accuracy of the three (5.2, 10.1 and 10.2)

ANN Model Modification with RMSProp Optimizer

[469]:
```
# model_ann_rmsprop = ANN_Creation(x_train=x_train, y_train_cat=y_train_cat)

# model_ann_rmsprop.compile(optimizer='rmsprop',
#                    loss='categorical_crossentropy',
#                    metrics=['accuracy'])

# model_ann_rmsprop.fit(x=x_train,
#               y=y_train_cat,
#               epochs=100,
#               batch_size=200,
#               validation_data=(x_test, y_test_cat),
#               verbose=2,
#               callbacks=[early_stop]
#              )

# loss_rmsprop = pd.DataFrame(model_ann_rmsprop.history.history)
# print(loss_rmsprop)
# loss_rmsprop[['accuracy','val_accuracy']].plot()
# loss_rmsprop[['loss','val_loss']].plot()
# plt.show()
```

[470]:
```
model_ann_rmsprop, loss_cv_rmsprop = ann_with_cross_val(x_train, y_train,␣
↪no_hiddenlayer=5, dropout_rate=0.0, k_fold=5,
                                                        optimizers ='rmsprop',␣
↪loss_val='categorical_crossentropy',
                                                        metrics_val='accuracy',␣
↪display_time = 'X')
```

```
Fold 1
Time taken for this fold: 55.064064502716064 seconds
Fold 2
Time taken for this fold: 30.318096160888672 seconds
Fold 3
Time taken for this fold: 37.78409123420715 seconds
Fold 4
```

```
Time taken for this fold: 36.599623680114746 seconds
Fold 5
Time taken for this fold: 41.23757004737854 seconds
Total time taken for training across all folds: 201.00344562530518 seconds
```

[471]:
```python
print(loss_cv_rmsprop)
```

```
       loss  accuracy  val_loss  val_accuracy
0  0.897853  0.808737  0.836942      0.809500
1  0.788156  0.825033  0.783983      0.823483
2  0.768002  0.827421  0.781574      0.823750
3  0.756655  0.828287  0.759172      0.825550
4  0.748279  0.829008  0.768535      0.820383
5  0.742412  0.829463  0.739856      0.831150
6  0.755554  0.827323  0.750969      0.833083
7  0.746685  0.828521  0.818654      0.798333
8  0.785092  0.826563  0.776527      0.820333
```

ANN Model Modification with ADAM Optimizer

[472]:
```python
# model_ann_adam = ANN_Creation(x_train=x_train, y_train_cat=y_train_cat)

# model_ann_adam.compile(optimizer='adam',
#                    loss='categorical_crossentropy',
#                    metrics=['accuracy'])

# model_ann_adam.fit(x=x_train,
#                y=y_train_cat,
#                epochs=100,
#                batch_size=200,
#                validation_data=(x_test, y_test_cat),
#                verbose=2,
#                callbacks=[early_stop]
#               )

# loss_adam = pd.DataFrame(model_ann_adam.history.history)
# print(loss_adam)
# loss_adam[['accuracy','val_accuracy']].plot()
# loss_adam[['loss','val_loss']].plot()
# plt.show()
```

[473]:
```python
model_ann_adam, loss_cv_adam = ann_with_cross_val(x_train, y_train,␣
 ↪no_hiddenlayer=5, dropout_rate=0.0,
                                                  k_fold=5, optimizers ='adam',␣
 ↪loss_val='categorical_crossentropy',
                                                  metrics_val='accuracy',␣
 ↪display_time = 'X')
```

```
Fold 1
Time taken for this fold: 94.23542809486389 seconds
Fold 2
Time taken for this fold: 24.31574845314026 seconds
Fold 3
Time taken for this fold: 21.595578908920288 seconds
Fold 4
Time taken for this fold: 36.678144693374634 seconds
Fold 5
Time taken for this fold: 26.9521541595459 seconds
Total time taken for training across all folds: 203.77705430984497 seconds
```

[474]:
```
print(loss_cv_adam)
```

```
        loss  accuracy  val_loss  val_accuracy
0   0.875889  0.821113  0.782294      0.830650
1   0.761543  0.835629  0.772860      0.827567
2   0.743086  0.837875  0.761269      0.830650
3   0.731179  0.838504  0.737148      0.832817
4   0.737841  0.836868  0.753807      0.829694
5   0.753838  0.836865  0.763967      0.833042
6   0.741227  0.838760  0.724720      0.844625
7   0.784606  0.837979  0.757779      0.842000
8   0.776179  0.835208  0.781743      0.834667
9   0.759780  0.837625  0.773998      0.832667
10  0.750369  0.836583  0.716523      0.851417
11  0.750025  0.840042  0.720180      0.845750
12  0.736473  0.838333  0.745204      0.841250
13  0.733527  0.840167  0.733549      0.838167
```

[ ]:
```

```

Plot Comparison of Training & Validation Accuracy of three Models(5.2, 10.1 and 10.2)

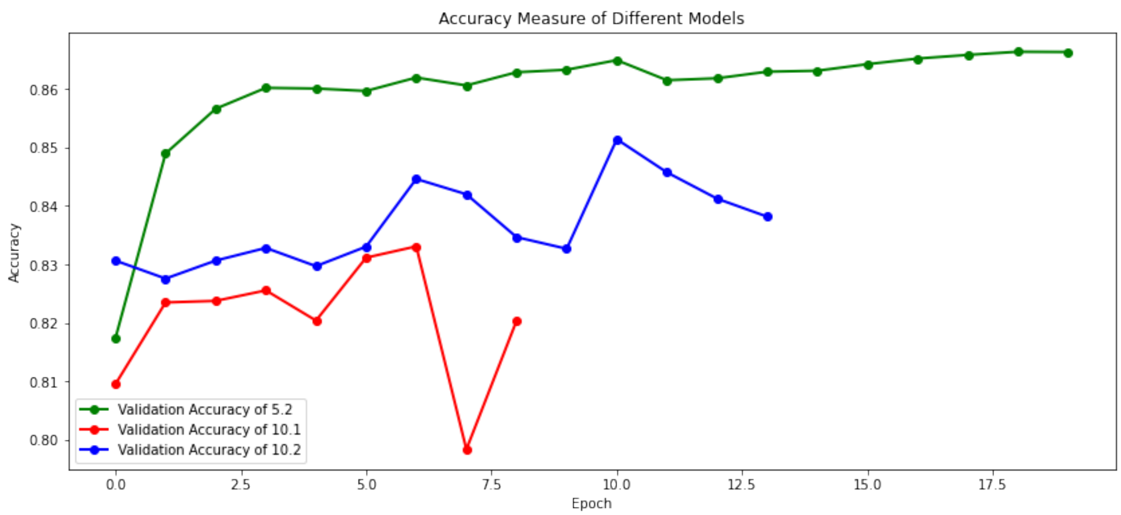
[475]:
```
loss_cv_sgd['index']=loss_cv_sgd.index
loss_cv_rmsprop['index']=loss_cv_rmsprop.index
loss_cv_adam['index']=loss_cv_adam.index
```

[476]:
```
plt.figure(figsize = (14, 6))
plt.plot(loss_cv_sgd.iloc[:,4],loss_cv_sgd.iloc[:,1], 'go-', label='Validation␣
 ↪Accuracy of 5.2', linewidth=2)
plt.plot(loss_cv_rmsprop.iloc[:,4],loss_cv_rmsprop.iloc[:,3], 'ro-',␣
 ↪label='Validation Accuracy of 10.1', linewidth=2)
plt.plot(loss_cv_adam.iloc[:,4],loss_cv_adam.iloc[:,3], 'bo-',␣
 ↪label='Validation Accuracy of 10.2', linewidth=2)

plt.title('Accuracy Measure of Different Models')
```

```
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



[477]:
```
am_max_5_2 = max(loss_cv_sgd.iloc[:,1])
am_max_10_1 = max(loss_cv_rmsprop.iloc[:,1])
am_max_10_2 = max(loss_cv_adam.iloc[:,1])
am_max = max(am_max_5_2,am_max_10_1, am_max_10_2)

if  am_max == am_max_5_2:
    print(f'Model 5.2 is having highest Validation Accuracy {am_max_5_2}')
elif am_max == am_max_8_1:
    print(f'Model 10.1 is having highest Validation Accuracy {am_max_10_1}')
else:
    print(f'Model 10.2 is having highest Validation Accuracy {am_max_10_2}')
```

```
Model 5.2 is having highest Validation Accuracy 0.8663958311080933
```

[ ]:
```

```

11 11. Conclusion - Score: 1 mark

Comparing the sections 4.1, 5.2, 8, 9, and 10, present your observations on which model or architecture or regualiser or optimiser perfomed better.

Below are designed models:

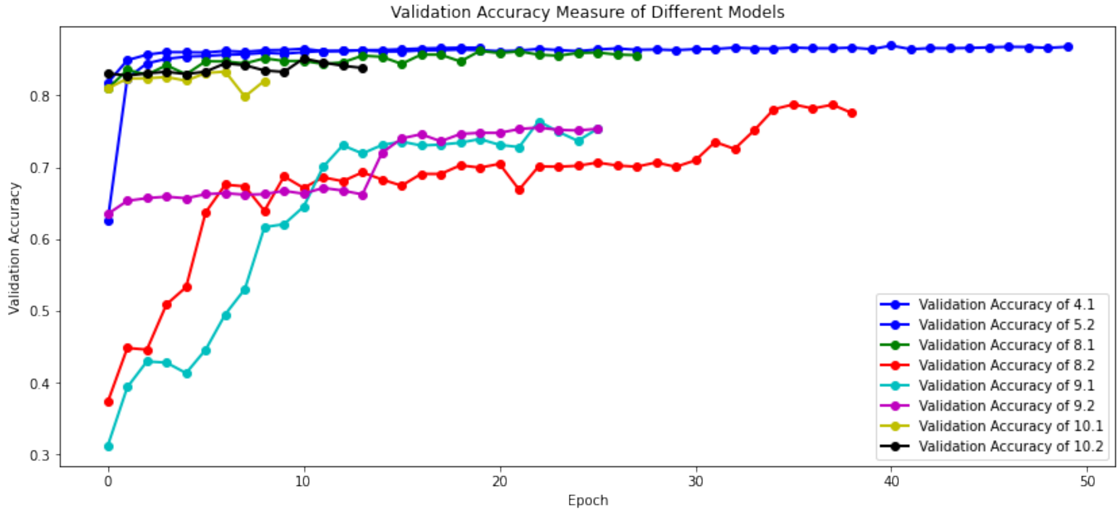
Model	Description
4.1	Simple ANN with ADAM optimizer & 5 hidden Layer
5.2	ANN with SGD optimozer (cross-validation)
8.1	Simple ANN with ADAM optimizer & 4 hidden Layer
8.2	Simple ANN with ADAM optimizer & 6 hidden Layer
9.1	Simple ANN with Dropout Ratio 0.25
9.2	Simple ANN with Dropout Ratio 0.25 & L2 regulariser 0.0001
10.1	ANN with RMSprop optimozer (cross-validation)
10.2	ANN with ADAM optimozer (cross-validation)

[478]:

```
plt.figure(figsize = (14, 6))
plt.plot(loss.iloc[:,4],loss.iloc[:,1], color='blue', marker='o',␣
 ↪label='Validation Accuracy of 4.1', linewidth=2)
plt.plot(loss_cv_sgd.iloc[:,4],loss_cv_sgd.iloc[:,1], color='blue', marker='o',␣
 ↪label='Validation Accuracy of 5.2', linewidth=2)
plt.plot(loss_minus.iloc[:,4],loss_minus.iloc[:,3], 'go-', label='Validation␣
 ↪Accuracy of 8.1', linewidth=2)
plt.plot(loss_plus.iloc[:,4],loss_plus.iloc[:,3], 'ro-', label='Validation␣
 ↪Accuracy of 8.2', linewidth=2)
plt.plot(loss_do.iloc[:,4],loss_do.iloc[:,3], 'co-', label='Validation Accuracy␣
 ↪of 9.1', linewidth=2)
plt.plot(loss_dor.iloc[:,4],loss_dor.iloc[:,3], 'mo-', label='Validation␣
 ↪Accuracy of 9.2', linewidth=2)
plt.plot(loss_cv_rmsprop.iloc[:,4],loss_cv_rmsprop.iloc[:,3], 'yo-',␣
 ↪label='Validation Accuracy of 10.1', linewidth=2)
plt.plot(loss_cv_adam.iloc[:,4],loss_cv_adam.iloc[:,3], 'ko-',␣
 ↪label='Validation Accuracy of 10.2', linewidth=2)

plt.title('Validation Accuracy Measure of Different Models')
plt.xlabel('Epoch')
plt.ylabel('Validation Accuracy')
plt.legend()

plt.show()
```



[479]:
```
am_max = max(am_max_4_1, am_max_5_2, am_max_8_1, am_max_8_2, am_max_9_1,␣
 ↪am_max_9_2, am_max_10_1, am_max_10_2)

if  am_max == am_max_4_1:
    print(f'Model 4.1 is having highest Validation Accuracy {am_max_4_1}')
elif  am_max == am_max_5_2:
    print(f'Model 5.2 is having highest Validation Accuracy {am_max_5_2}')
elif am_max == am_max_8_1:
    print(f'Model 8.1 is having highest Validation Accuracy {am_max_8_1}')
elif am_max == am_max_8_2:
    print(f'Model 8.2 is having highest Validation Accuracy {am_max_8_2}')
elif am_max == am_max_9_1:
    print(f'Model 9.1 is having highest Validation Accuracy {am_max_9_1}')
elif am_max == am_max_9_2:
    print(f'Model 9.2 is having highest Validation Accuracy {am_max_9_2}')
elif am_max == am_max_10_1:
    print(f'Model 10.1 is having highest Validation Accuracy {am_max_10_1}')
elif am_max == am_max_10_2:
    print(f'Model 10.2 is having highest Validation Accuracy {am_max_10_2}')
```

```
Model 8.1 is having highest Validation Accuracy 0.8701000213623047
```

Comparison of Models for Loss & Accuracy wrt Testset

[480]:
```
score_ann = model_ann.evaluate(x=x_test, y=y_test_cat, verbose = 0)
score_sgd = model_ann_sgd.evaluate(x=x_test, y=y_test_cat, verbose = 0)
score_minus = model_ann_minus.evaluate(x=x_test, y=y_test_cat, verbose = 0)
score_plus = model_ann_plus.evaluate(x=x_test, y=y_test_cat, verbose = 0)
score_do = model_ann_do.evaluate(x=x_test, y=y_test_cat, verbose = 0)
score_dor = model_ann_dor.evaluate(x=x_test, y=y_test_cat, verbose = 0)
```

```
score_rmsprop = model_ann_rmsprop.evaluate(x=x_test, y=y_test_cat, verbose = 0)
score_adam = model_ann_adam.evaluate(x=x_test, y=y_test_cat, verbose = 0)
```

[481]:
```
scores_loss = [score_ann[0], score_sgd[0], score_minus[0], score_plus[0], ↪score_do[0], score_dor[0],
               score_rmsprop[0], score_adam[0]]
```

[482]:
```
models = ['4.1', '5.2', '8.1', '8.2', '9.1', '9.2', '10.1', '10.2']
```

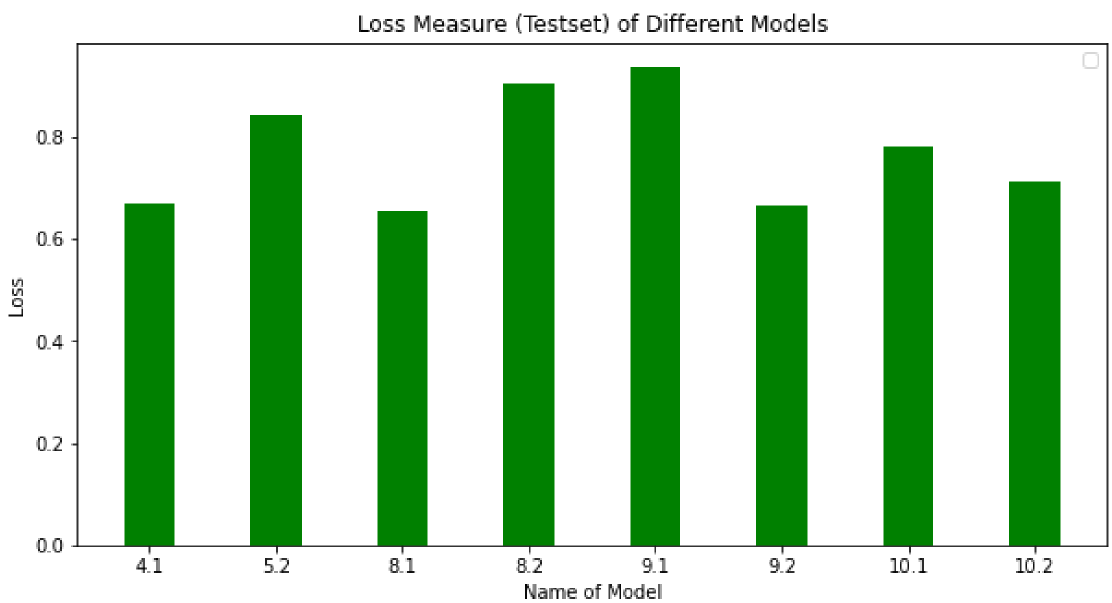
[483]:
```
plt.figure(figsize = (10, 5))
plt.bar(models, scores_loss, color ='green', width = 0.4)

# for i, v in enumerate(scores):
#     plt.text(v, i, str(v),
#             color = 'blue', fontweight = 'bold')

plt.title('Loss Measure (Testset) of Different Models')
plt.xlabel('Name of Model')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label
start with an underscore are ignored when legend() is called with no argument.
```



[484]:
```
print(f'Lowest Loss Value of All Models for Testing Set = {min(scores_loss)}')
```

```
Lowest Loss Value of All Models for Testing Set = 0.6545193195343018
```

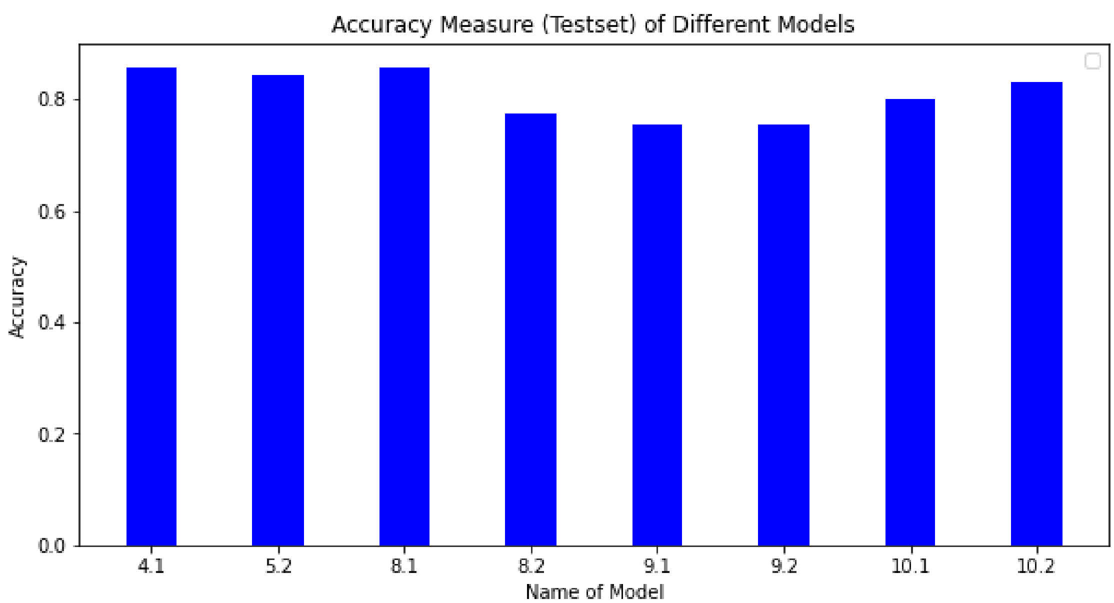
[485]:
```python
scores_accuracy = [score_ann[1], score_sgd[1], score_minus[1], score_plus[1],
                   score_do[1], score_dor[1], score_rmsprop[1], score_adam[1]]
```

[486]:
```python
plt.figure(figsize = (10, 5))
plt.bar(models, scores_accuracy, color ='blue', width = 0.4)

plt.title('Accuracy Measure (Testset) of Different Models')
plt.xlabel('Name of Model')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label
start with an underscore are ignored when legend() is called with no argument.
```



[487]:
```python
print(f'Highest Accuracy Value of All Models for Testing Set =
 ↪{max(scores_accuracy)}')
```

```
Highest Accuracy Value of All Models for Testing Set = 0.857200026512146
```

Alternative Way of Hyper-Parameter Tuning via KerasTuner

- KerasTuner is an easy-to-use, scalable hyperparameter optimization framework that solves pain points of hyperparameter search.
- Search Space can easily be configured with a define-by-run syntax, then leverage one of the available search algorithms to find best hyperparameter values for models.

[488]:
```
def build_model(hp):  # random search passes this hyperparameter() object
    model = Sequential()

    activation = hp.Choice(name = 'activation', values = ['relu', 'tanh',␣
 ↪'sigmoid'], ordered = False, default='relu')
#     dropout = hp.Boolean(name = 'dropout', default = False)
#     learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])
    no_layer = hp.Int(name = 'no_layer', min_value = 2, max_value = 10)
    no_unit = hp.Int(f'layer_unit', min_value = 32, max_value = 480, step=32,␣
 ↪default=32)
    optimizer = hp.Choice(name = 'optimizer', values = ['adam', 'sgd',␣
 ↪'rmsprop'], default='adam')

    #Input layer
    model.add(Flatten(input_shape=(x_train.shape[1], x_train.shape[2], x_train.
 ↪shape[3])))


    for i in range(no_layer):
        model.add(Dense(no_unit, activation=activation))
#         model.add(Dropout(0.2))


    #Output Layer
    model.add(Dense(units=y_train_cat.shape[1], activation='softmax'))

    model.compile(optimizer=optimizer,
#                   loss='binary_crossentropy',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])


    return model
```

[489]:
```
LOG_DIR = f"{int(time.time())}"

tuner = RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=5,  # how many model variations to test?
    executions_per_trial=5,  # how many trials per variation? (same model could␣
 ↪perform differently)
    directory=LOG_DIR)
```

[490]:
```
tuner.search_space_summary()
```

```
Search space summary
```

```
Default search space size: 4
activation (Choice)
{'default': 'relu', 'conditions': [], 'values': ['relu', 'tanh', 'sigmoid'],
'ordered': False}
no_layer (Int)
{'default': None, 'conditions': [], 'min_value': 2, 'max_value': 10, 'step': 1,
'sampling': 'linear'}
layer_unit (Int)
{'default': 32, 'conditions': [], 'min_value': 32, 'max_value': 480, 'step': 32,
'sampling': 'linear'}
optimizer (Choice)
{'default': 'adam', 'conditions': [], 'values': ['adam', 'sgd', 'rmsprop'],
'ordered': False}
```

[491]:
```
tuner.search(x=x_train,
              y=y_train_cat,
              epochs=1,
              batch_size=64,
              validation_data=(x_test, y_test_cat),
              verbose=2,
#               callbacks=[early_stop]
             )
```

```
Trial 5 Complete [00h 00m 30s]
val_accuracy: 0.8265399932861328

Best val_accuracy So Far: 0.8265399932861328
Total elapsed time: 00h 02m 10s
INFO:tensorflow:Oracle triggered exit
```

[492]:
```
tuner.results_summary()
```

```
Results summary
Results in 1687637311\untitled_project
Showing 10 best trials
Objective(name="val_accuracy", direction="max")

Trial 4 summary
Hyperparameters:
activation: sigmoid
no_layer: 2
layer_unit: 384
optimizer: rmsprop
Score: 0.8265399932861328

Trial 3 summary
Hyperparameters:
activation: tanh
```

```
no_layer: 6
layer_unit: 128
optimizer: rmsprop
Score: 0.8246399879455566

Trial 1 summary
Hyperparameters:
activation: tanh
no_layer: 3
layer_unit: 352
optimizer: rmsprop
Score: 0.8206399917602539

Trial 2 summary
Hyperparameters:
activation: relu
no_layer: 4
layer_unit: 192
optimizer: sgd
Score: 0.7786999940872192

Trial 0 summary
Hyperparameters:
activation: sigmoid
no_layer: 3
layer_unit: 32
optimizer: rmsprop
Score: 0.7274199962615967
```

[493]:
```
print(tuner.get_best_hyperparameters()[0].values)
```

```
{'activation': 'sigmoid', 'no_layer': 2, 'layer_unit': 384, 'optimizer':
'rmsprop'}
```

[494]:
```
best_model = tuner.get_best_models()[0]
print(best_model.summary())
```

```
Model: "sequential"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 384)               301440

 dense_1 (Dense)             (None, 384)               147840

 dense_2 (Dense)             (None, 10)                3850
```

```
=================================================================
Total params: 453,130
Trainable params: 453,130
Non-trainable params: 0
_________________________________________________________________
None
```

[495]:
```
best_model.fit(x=x_train,
              y=y_train_cat,
              epochs=200,
              batch_size=200,
              validation_data=(x_test, y_test_cat),
              verbose=2,
              callbacks=[early_stop]
             )
```

```
Epoch 1/200
300/300 - 2s - loss: 0.4004 - accuracy: 0.8531 - val_loss: 0.4417 -
val_accuracy: 0.8333 - 2s/epoch - 7ms/step
Epoch 2/200
300/300 - 1s - loss: 0.3733 - accuracy: 0.8638 - val_loss: 0.4136 -
val_accuracy: 0.8501 - 1s/epoch - 5ms/step
Epoch 3/200
300/300 - 2s - loss: 0.3549 - accuracy: 0.8699 - val_loss: 0.4234 -
val_accuracy: 0.8466 - 2s/epoch - 7ms/step
Epoch 4/200
300/300 - 2s - loss: 0.3364 - accuracy: 0.8764 - val_loss: 0.3686 -
val_accuracy: 0.8661 - 2s/epoch - 7ms/step
Epoch 5/200
300/300 - 2s - loss: 0.3232 - accuracy: 0.8808 - val_loss: 0.3817 -
val_accuracy: 0.8619 - 2s/epoch - 7ms/step
Epoch 6/200
300/300 - 2s - loss: 0.3086 - accuracy: 0.8856 - val_loss: 0.3789 -
val_accuracy: 0.8595 - 2s/epoch - 8ms/step
Epoch 7/200
300/300 - 2s - loss: 0.2980 - accuracy: 0.8899 - val_loss: 0.3429 -
val_accuracy: 0.8762 - 2s/epoch - 7ms/step
Epoch 8/200
300/300 - 2s - loss: 0.2896 - accuracy: 0.8923 - val_loss: 0.3589 -
val_accuracy: 0.8703 - 2s/epoch - 6ms/step
Epoch 9/200
300/300 - 2s - loss: 0.2796 - accuracy: 0.8951 - val_loss: 0.3425 -
val_accuracy: 0.8792 - 2s/epoch - 6ms/step
Epoch 10/200
300/300 - 2s - loss: 0.2719 - accuracy: 0.8987 - val_loss: 0.3332 -
val_accuracy: 0.8802 - 2s/epoch - 6ms/step
Epoch 11/200
```

```
300/300 - 2s - loss: 0.2636 - accuracy: 0.9022 - val_loss: 0.3337 -
val_accuracy: 0.8808 - 2s/epoch - 6ms/step
Epoch 12/200
300/300 - 2s - loss: 0.2557 - accuracy: 0.9037 - val_loss: 0.3297 -
val_accuracy: 0.8822 - 2s/epoch - 6ms/step
Epoch 13/200
300/300 - 2s - loss: 0.2486 - accuracy: 0.9067 - val_loss: 0.3373 -
val_accuracy: 0.8807 - 2s/epoch - 7ms/step
Epoch 14/200
300/300 - 2s - loss: 0.2432 - accuracy: 0.9079 - val_loss: 0.3280 -
val_accuracy: 0.8811 - 2s/epoch - 7ms/step
Epoch 15/200
300/300 - 2s - loss: 0.2378 - accuracy: 0.9103 - val_loss: 0.3383 -
val_accuracy: 0.8836 - 2s/epoch - 8ms/step
Epoch 16/200
300/300 - 2s - loss: 0.2294 - accuracy: 0.9136 - val_loss: 0.3288 -
val_accuracy: 0.8885 - 2s/epoch - 7ms/step
Epoch 17/200
300/300 - 2s - loss: 0.2237 - accuracy: 0.9155 - val_loss: 0.3305 -
val_accuracy: 0.8837 - 2s/epoch - 7ms/step
```

[495]: <keras.callbacks.History at 0x1c19ea293a0>

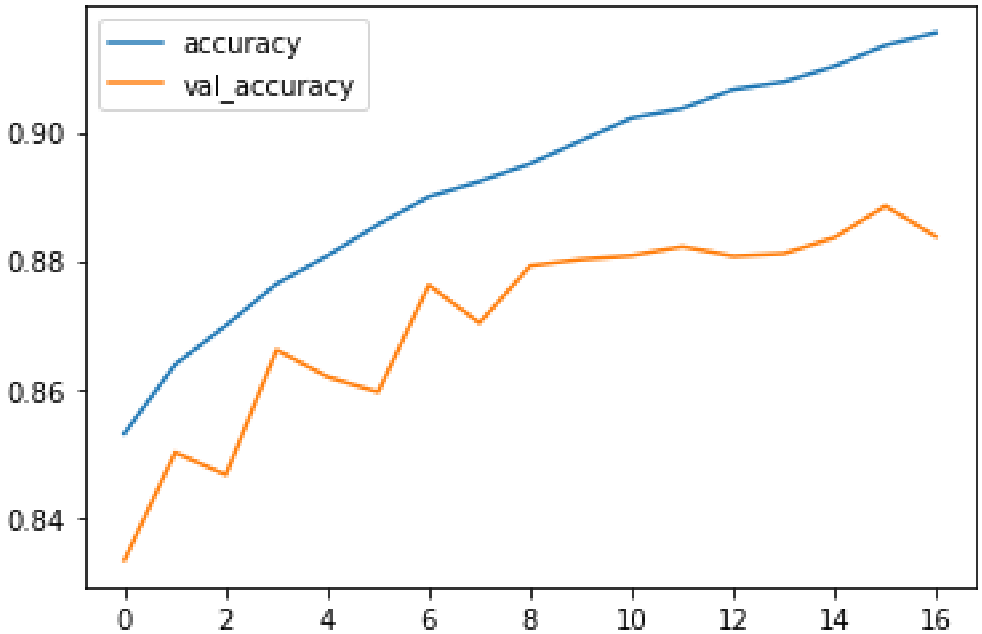
[496]:
```
loss_best = pd.DataFrame(best_model.history.history)
```

[497]:
```
print(loss_best)
```

```
        loss  accuracy  val_loss  val_accuracy
0   0.400376  0.853067  0.441706        0.8333
1   0.373308  0.863817  0.413570        0.8501
2   0.354889  0.869900  0.423449        0.8466
3   0.336384  0.876400  0.368647        0.8661
4   0.323188  0.880767  0.381677        0.8619
5   0.308609  0.885600  0.378858        0.8595
6   0.298041  0.889933  0.342871        0.8762
7   0.289569  0.892317  0.358927        0.8703
8   0.279559  0.895100  0.342497        0.8792
9   0.271918  0.898683  0.333231        0.8802
10  0.263604  0.902217  0.333689        0.8808
11  0.255730  0.903733  0.329749        0.8822
12  0.248574  0.906650  0.337292        0.8807
13  0.243221  0.907850  0.327977        0.8811
14  0.237797  0.910300  0.338254        0.8836
15  0.229402  0.913567  0.328834        0.8885
16  0.223704  0.915533  0.330454        0.8837
```

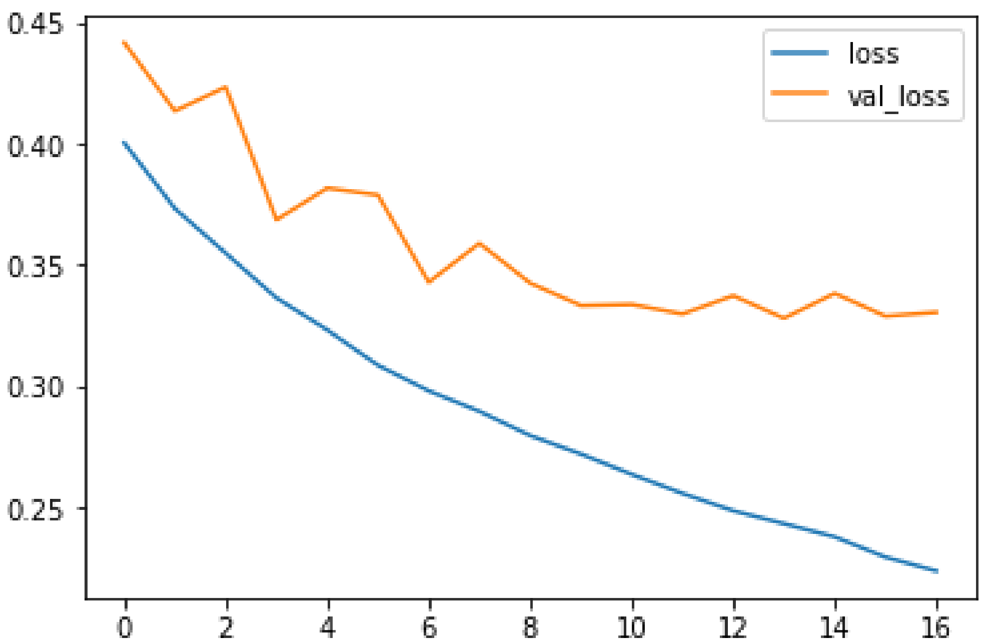
[498]:
```
loss_best[['accuracy','val_accuracy']].plot()
```

[498]: <AxesSubplot:>



[499]:
```
loss_best[['loss','val_loss']].plot()
```

[499]: <AxesSubplot:>



[500]:
```
# print(best_model.metrics_names)
score_best = best_model.evaluate(x=x_test, y=y_test_cat, verbose = 0)
```

[501]:
```
score_best
```

[501]:
```
[0.3304544687271118, 0.8837000131607056]
```

[502]:
```
print(f'For Testset, Best Optimized model via Tuner has {best_model.
↪metrics_names[0]} = {score_best[0]} & {best_model.metrics_names[1]} =␣
↪{score_best[1]}')
```

```
For Testset, Best Optimized model via Tuner has loss = 0.3304544687271118 &
accuracy = 0.8837000131607056
```

11.0.1 NOTE

All Late Submissions will incur a penalty of -2 marks . So submit your assignments on time.

Good Luck