FASHION MNIST CLASSIFICATION

Abstract:

This report focuses on the performance of several classifiers on the Fashion-MNIST dataset. Fashion-MNIST is a more complex image dataset. The data is normalized and principal component analysis are applied. MPP cases 1, 2, and 3, k-nearest neighbors, and Sklearn are evaluated on the dataset. Additionally, 3-layer backpropagation neural networks and a convolutional neural network (CNN) are also tested. Performance for these classifiers is compared using the data's built-in train/test split and using 10-fold cross validation. Additionally, k-means and winner-takes-all clustering techniques are investigated for visualizing and reproducing the clusters in the data.

Objective:

The objective is to identify (predict) different fashion products from the given images using various best possible Machine Learning Models (Algorithms) and compare their results (performance measures/scores) to arrive at the best ML model.

Introduction:

The Fashion-MNIST clothing classification problem is a new standard dataset used in computer vision and deep learning.Although the dataset is relatively simple, it can be used as the basis for learning and practicing how to develop, evaluate, and use deep convolutional neural networks for image classification from scratch. This includes how to develop a robust test harness for estimating the performance of the model, how to explore improvements to the model, and how to save the model and later load it to make predictions on new data.

Methodology:

The Fashion MNIST dataset was developed as a response to the wide use of the MNIST dataset, that has been effectively "*solved*" given the use of modern convolutional neural networks.

Convolutional Neural Networks (CNN):

Convolutional neural networks are formed mainly from three types of layers: convolutional, max pooling, and fully-connected layers. The first stages of convolutional nets consist of max pooling and convolutional layers. First, convolutional layers attempt to extract features by sliding a filter over the previous layer. Next, max pooling is used to merge semantically related features together. These layers are often used together in order to extract more and more complex features from the original image. After convolutional and max pooling layers have been applied, fully-connected layers are used to predict the class of the data sample. In this project, Keras is used to implement the CNN.

k-Nearest Neighbor:

k-Nearest Neighbor (kNN) learning is one of the commonly used methods in supervised learning. The mechanism can be easily understood. For a given testing set, k nearest training samples will be selected, and predictions will be based on these k "neighbors" by voting or averaging in general. k-Nearest Neighbor is also known as a method of lazy learning, which has no running time cost in the training stage. It saves the training samples and processes them in the later stage. Although kNN algorithm is simple, the risk of kNN is still less than 2 times of optimal Bayes risk.

Backpropagation Neural Network:

A 3-layer BPNN (Backpropagation Neural Network) is a neural network architecture that consists of three layers of neurons: an input layer, a hidden layer, and an output layer.

The input layer receives the input values and passes them through to the hidden layer, where the values are transformed by a set of weights and biases. The hidden layer applies an activation function to the weighted sum of inputs and biases, and outputs the results to the output layer. The output layer then applies another activation function to produce the final output values.

CODE:

working with a larger example

In [2]:

```
import tensorflow as tf
from tensorflow.keras.datasets import fashion_mnist

(train_data , train_labels), (test_data , test_labels) =
fashion_mnist.load_data()
```

viewing first few training examples

In [34]:

```
print(f"Training sample:\n{train_data[0]}\n")
print(f"Training label: {train_labels[0]}")
```

```
Training sample:
[[0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.00392157 0.         0.         0.05098039 0.28627451 0.
  0.         0.00392157 0.01568627 0.         0.         0.
  0.         0.00392157 0.00392157 0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.01176471 0.         0.14117647 0.53333333 0.49803922 0.24313725
```

```
 0.21176471 0.         0.         0.         0.00392157 0.01176471
 0.01568627 0.         0.         0.01176471]
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.02352941 0.         0.4        0.8        0.69019608 0.5254902
 0.56470588 0.48235294 0.09019608 0.         0.         0.
 0.         0.04705882 0.03921569 0.        ]
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.60784314 0.9254902  0.81176471 0.69803922
 0.41960784 0.61176471 0.63137255 0.42745098 0.25098039 0.09019608
 0.30196078 0.50980392 0.28235294 0.05882353]
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.00392157
 0.         0.27058824 0.81176471 0.8745098  0.85490196 0.84705882
 0.84705882 0.63921569 0.49803922 0.4745098  0.47843137 0.57254902
 0.55294118 0.34509804 0.6745098  0.25882353]
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.00392157 0.00392157 0.00392157
 0.         0.78431373 0.90980392 0.90980392 0.91372549 0.89803922
 0.8745098  0.8745098  0.84313725 0.83529412 0.64313725 0.49803922
 0.48235294 0.76862745 0.89803922 0.        ]
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.71764706 0.88235294 0.84705882 0.8745098  0.89411765
 0.92156863 0.89019608 0.87843137 0.87058824 0.87843137 0.86666667
 0.8745098  0.96078431 0.67843137 0.        ]
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.75686275 0.89411765 0.85490196 0.83529412 0.77647059
 0.70588235 0.83137255 0.82352941 0.82745098 0.83529412 0.8745098
 0.8627451  0.95294118 0.79215686 0.        ]
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.00392157 0.01176471 0.
 0.04705882 0.85882353 0.8627451  0.83137255 0.85490196 0.75294118
 0.6627451  0.89019608 0.81568627 0.85490196 0.87843137 0.83137255
 0.88627451 0.77254902 0.81960784 0.20392157]
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.02352941 0.
 0.38823529 0.95686275 0.87058824 0.8627451  0.85490196 0.79607843
 0.77647059 0.86666667 0.84313725 0.83529412 0.87058824 0.8627451
 0.96078431 0.46666667 0.65490196 0.21960784]
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.01568627 0.         0.
 0.21568627 0.9254902  0.89411765 0.90196078 0.89411765 0.94117647
 0.90980392 0.83529412 0.85490196 0.8745098  0.91764706 0.85098039
 0.85098039 0.81960784 0.36078431 0.        ]
[0.         0.         0.00392157 0.01568627 0.02352941 0.02745098
 0.00784314 0.         0.         0.         0.         0.
 0.92941176 0.88627451 0.85098039 0.8745098  0.87058824 0.85882353
 0.87058824 0.86666667 0.84705882 0.8745098  0.89803922 0.84313725
 0.85490196 1.         0.30196078 0.        ]
[0.         0.01176471 0.         0.         0.         0.
 0.         0.         0.         0.24313725 0.56862745 0.8
 0.89411765 0.81176471 0.83529412 0.86666667 0.85490196 0.81568627
 0.82745098 0.85490196 0.87843137 0.8745098  0.85882353 0.84313725
 0.87843137 0.95686275 0.62352941 0.        ]
```

```
[0.         0.         0.         0.         0.07058824 0.17254902
 0.32156863 0.41960784 0.74117647 0.89411765 0.8627451  0.87058824
 0.85098039 0.88627451 0.78431373 0.80392157 0.82745098 0.90196078
 0.87843137 0.91764706 0.69019608 0.7372549  0.98039216 0.97254902
 0.91372549 0.93333333 0.84313725 0.        ]
[0.         0.22352941 0.73333333 0.81568627 0.87843137 0.86666667
 0.87843137 0.81568627 0.8        0.83921569 0.81568627 0.81960784
 0.78431373 0.62352941 0.96078431 0.75686275 0.80784314 0.8745098
 1.         1.         0.86666667 0.91764706 0.86666667 0.82745098
 0.8627451  0.90980392 0.96470588 0.        ]
[0.01176471 0.79215686 0.89411765 0.87843137 0.86666667 0.82745098
 0.82745098 0.83921569 0.80392157 0.80392157 0.80392157 0.8627451
 0.94117647 0.31372549 0.58823529 1.         0.89803922 0.86666667
 0.7372549  0.60392157 0.74901961 0.82352941 0.8        0.81960784
 0.87058824 0.89411765 0.88235294 0.        ]
[0.38431373 0.91372549 0.77647059 0.82352941 0.87058824 0.89803922
 0.89803922 0.91764706 0.97647059 0.8627451  0.76078431 0.84313725
 0.85098039 0.94509804 0.25490196 0.28627451 0.41568627 0.45882353
 0.65882353 0.85882353 0.86666667 0.84313725 0.85098039 0.8745098
 0.8745098  0.87843137 0.89803922 0.11372549]
[0.29411765 0.8        0.83137255 0.8        0.75686275 0.80392157
 0.82745098 0.88235294 0.84705882 0.7254902  0.77254902 0.80784314
 0.77647059 0.83529412 0.94117647 0.76470588 0.89019608 0.96078431
 0.9372549  0.8745098  0.85490196 0.83137255 0.81960784 0.87058824
 0.8627451  0.86666667 0.90196078 0.2627451 ]
[0.18823529 0.79607843 0.71764706 0.76078431 0.83529412 0.77254902
 0.7254902  0.74509804 0.76078431 0.75294118 0.79215686 0.83921569
 0.85882353 0.86666667 0.8627451  0.9254902  0.88235294 0.84705882
 0.78039216 0.80784314 0.72941176 0.70980392 0.69411765 0.6745098
 0.70980392 0.80392157 0.80784314 0.45098039]
[0.         0.47843137 0.85882353 0.75686275 0.70196078 0.67058824
 0.71764706 0.76862745 0.8        0.82352941 0.83529412 0.81176471
 0.82745098 0.82352941 0.78431373 0.76862745 0.76078431 0.74901961
 0.76470588 0.74901961 0.77647059 0.75294118 0.69019608 0.61176471
 0.65490196 0.69411765 0.82352941 0.36078431]
[0.         0.         0.29019608 0.74117647 0.83137255 0.74901961
 0.68627451 0.6745098  0.68627451 0.70980392 0.7254902  0.7372549
 0.74117647 0.7372549  0.75686275 0.77647059 0.8        0.81960784
 0.82352941 0.82352941 0.82745098 0.7372549  0.7372549  0.76078431
 0.75294118 0.84705882 0.66666667 0.        ]
[0.00784314 0.         0.         0.         0.25882353 0.78431373
 0.87058824 0.92941176 0.9372549  0.94901961 0.96470588 0.95294118
 0.95686275 0.86666667 0.8627451  0.75686275 0.74901961 0.70196078
 0.71372549 0.71372549 0.70980392 0.69019608 0.65098039 0.65882353
 0.38823529 0.22745098 0.         0.        ]
[0.         0.         0.         0.         0.         0.
 0.         0.15686275 0.23921569 0.17254902 0.28235294 0.16078431
 0.1372549  0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.        ]
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.        ]
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
```

```
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.        ]]
```

```
Training label: 9
```

In [5]:

```
train_data.shape , train_labels.shape , test_data.shape , test_labels.shape
```

Out[5]:

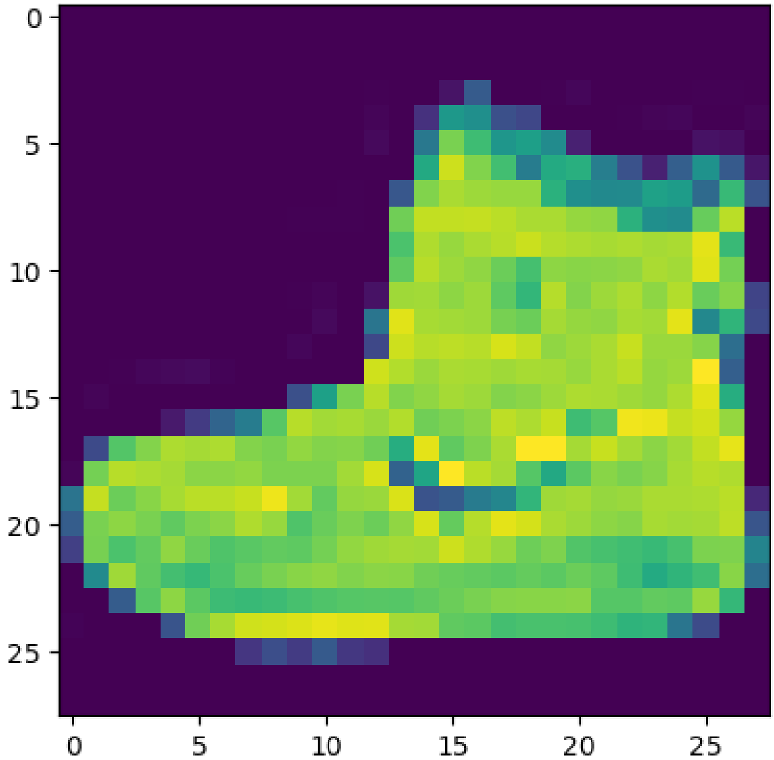
```
((60000, 28, 28), (60000,), (10000, 28, 28), (10000,))
```

In [6]:

```
# plot a single example
```

In [7]:

```
import matplotlib.pyplot as plt
plt.imshow(train_data[0]);
```



In [8]:

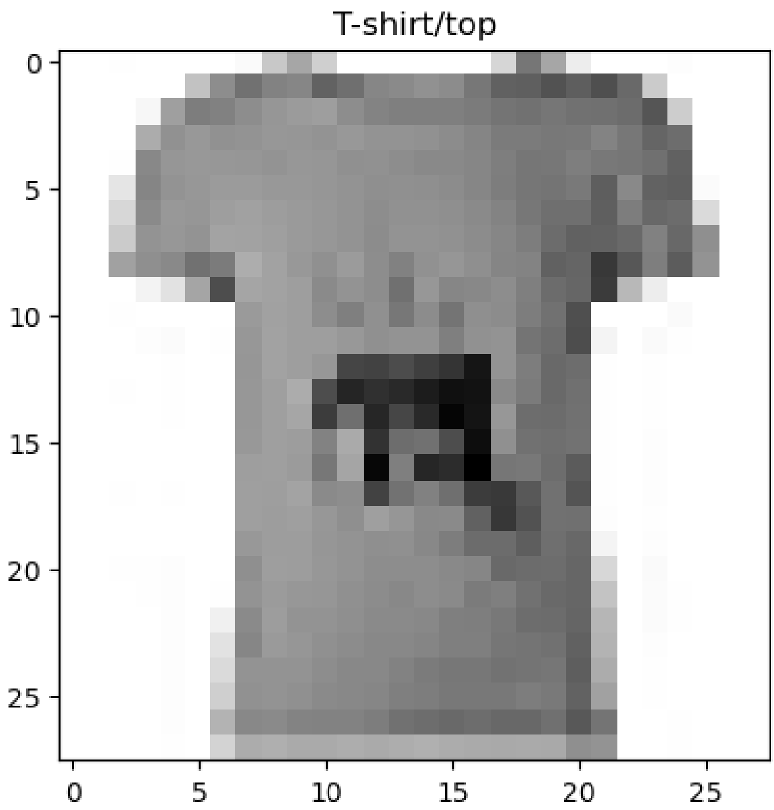
```
train_labels[0]

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

len(class_names)
```

Out[8]:

```
10
```

In [9]:

```
#plot an example image and its label

plt.imshow(train_data[180], cmap = plt.cm.binary)
plt.title(class_names[train_labels[180]]);
```



In [10]:

```
# plot multiple random images of fashion MNIST

import random
plt.figure(figsize =(7,7))
for i in range(4):
    ax = plt.subplot(2,2, i+1)
    rand_index = random.choice(range(len(train_data)))
    plt.imshow(train_data[rand_index], cmap=plt.cm.binary)
    plt.title(class_names[train_labels[rand_index]])
    plt.axis(False)
```

In [11]:

```
tf.random.set_seed(42)

model_1 = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(4, activation = "relu"),
    tf.keras.layers.Dense(4, activation = "relu"),
    tf.keras.layers.Dense(10, activation = "softmax") #output shape is 10
])

model_1.compile(loss = tf.keras.losses.SparseCategoricalCrossentropy(),
                optimizer = tf.keras.optimizers.Adam(),
                metrics = ['accuracy'])

non_norm_history = model_1.fit(train_data,
                               train_labels,
                               epochs=10,
                               validation_data = (test_data,
test_labels))
```

```
Epoch 1/10
1875/1875 [==============================] - 2s 990us/step - loss: 2.1858 -
accuracy: 0.1628 - val_loss: 1.7542 - val_accuracy: 0.2900
```

```
Epoch 2/10
1875/1875 [==============================] - 3s 1ms/step - loss: 1.5624 - a
ccuracy: 0.3524 - val_loss: 1.4271 - val_accuracy: 0.4231
Epoch 3/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.3597 - a
ccuracy: 0.4343 - val_loss: 1.3321 - val_accuracy: 0.4450
Epoch 4/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.3068 - a
ccuracy: 0.4476 - val_loss: 1.2793 - val_accuracy: 0.4570
Epoch 5/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.2664 - a
ccuracy: 0.4577 - val_loss: 1.2755 - val_accuracy: 0.4537
Epoch 6/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.2495 - a
ccuracy: 0.4600 - val_loss: 1.2440 - val_accuracy: 0.4655
Epoch 7/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.2350 - a
ccuracy: 0.4643 - val_loss: 1.2627 - val_accuracy: 0.4696
Epoch 8/10
1875/1875 [==============================] - 5s 3ms/step - loss: 1.1656 - a
ccuracy: 0.5120 - val_loss: 1.0846 - val_accuracy: 0.5359
Epoch 9/10
1875/1875 [==============================] - 5s 3ms/step - loss: 1.0315 - a
ccuracy: 0.5644 - val_loss: 1.0795 - val_accuracy: 0.5487
Epoch 10/10
1875/1875 [==============================] - 5s 3ms/step - loss: 1.0086 - a
ccuracy: 0.5750 - val_loss: 1.0174 - val_accuracy: 0.5833
```

In [12]:

```
model_1.evaluate(test_data, test_labels)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 1.0174 - acc
uracy: 0.5833
```

Out[12]:

```
[1.0173819065093994, 0.583299994468689]
```

In [13]:

```
train_data.min() , train_data.max()
```

Out[13]:

```
(0, 255)
```

In [14]:

```
train_data = train_data/255.0
test_data = test_data/255.0
```

In [15]:

```
train_data.min() , train_data.max()
```

Out[15]:

```
(0.0, 1.0)
```

In [16]:

```
tf.random.set_seed(42)

model_2 = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28)),
    tf.keras.layers.Dense(4, activation = "relu"),
    tf.keras.layers.Dense(4, activation = "relu"),
    tf.keras.layers.Dense(10, activation = "softmax")
])
```

```
model_2.compile(loss = tf.keras.losses.SparseCategoricalCrossentropy(),
                optimizer = tf.keras.optimizers.Adam(),
                metrics = ["accuracy"])

norm_history = model_2.fit(train_data,
                           train_labels,
                           epochs=10,
                           validation_data = (test_data , test_labels))
```

```
Epoch 1/10
1875/1875 [==============================] - 6s 3ms/step - loss: 1.0941 - a
ccuracy: 0.6042 - val_loss: 0.8030 - val_accuracy: 0.7186
Epoch 2/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6944 - a
ccuracy: 0.7587 - val_loss: 0.6742 - val_accuracy: 0.7682
Epoch 3/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6182 - a
ccuracy: 0.7904 - val_loss: 0.6225 - val_accuracy: 0.7869
Epoch 4/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5785 - a
ccuracy: 0.8044 - val_loss: 0.5982 - val_accuracy: 0.8036
Epoch 5/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5565 - a
ccuracy: 0.8122 - val_loss: 0.5795 - val_accuracy: 0.8057
Epoch 6/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5392 - a
ccuracy: 0.8183 - val_loss: 0.5973 - val_accuracy: 0.7996
Epoch 7/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5291 - a
ccuracy: 0.8226 - val_loss: 0.5716 - val_accuracy: 0.8045
Epoch 8/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5184 - a
ccuracy: 0.8252 - val_loss: 0.5689 - val_accuracy: 0.8089
Epoch 9/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5126 - a
ccuracy: 0.8270 - val_loss: 0.5681 - val_accuracy: 0.8149
Epoch 10/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5074 - a
ccuracy: 0.8294 - val_loss: 0.5450 - val_accuracy: 0.8189
```

In [17]:

```
model_2.evaluate(test_data , test_labels)
```

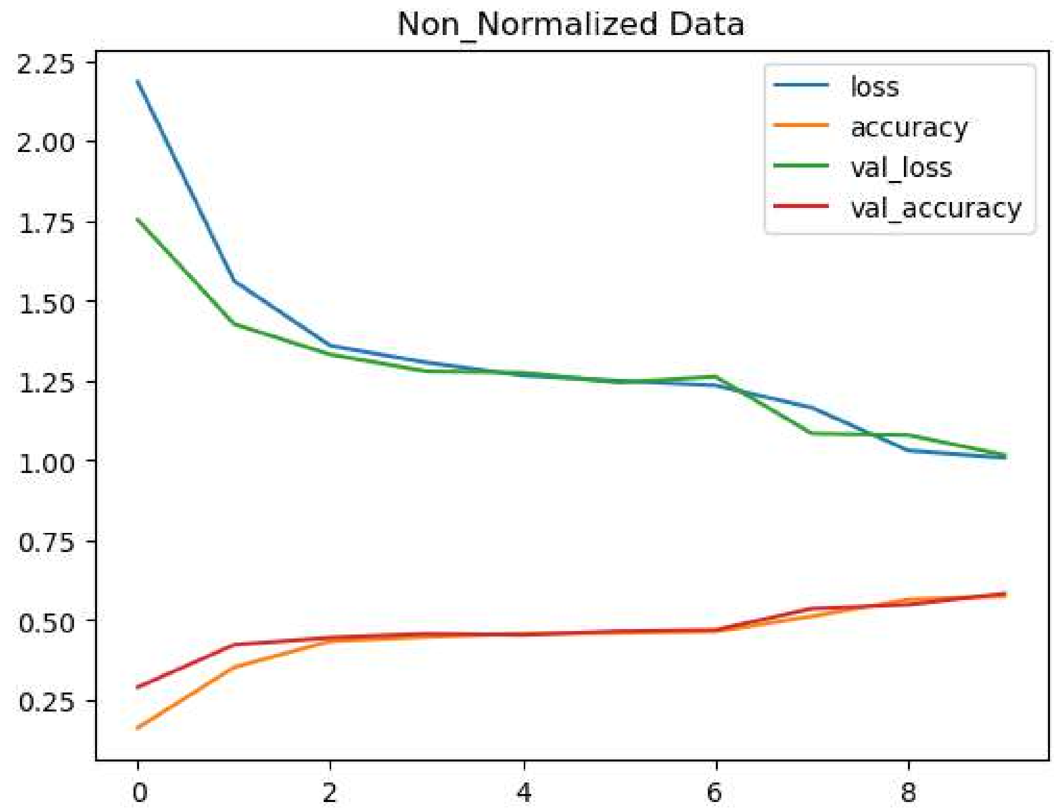
```
313/313 [==============================] - 1s 2ms/step - loss: 0.5450 - acc
uracy: 0.8189
```

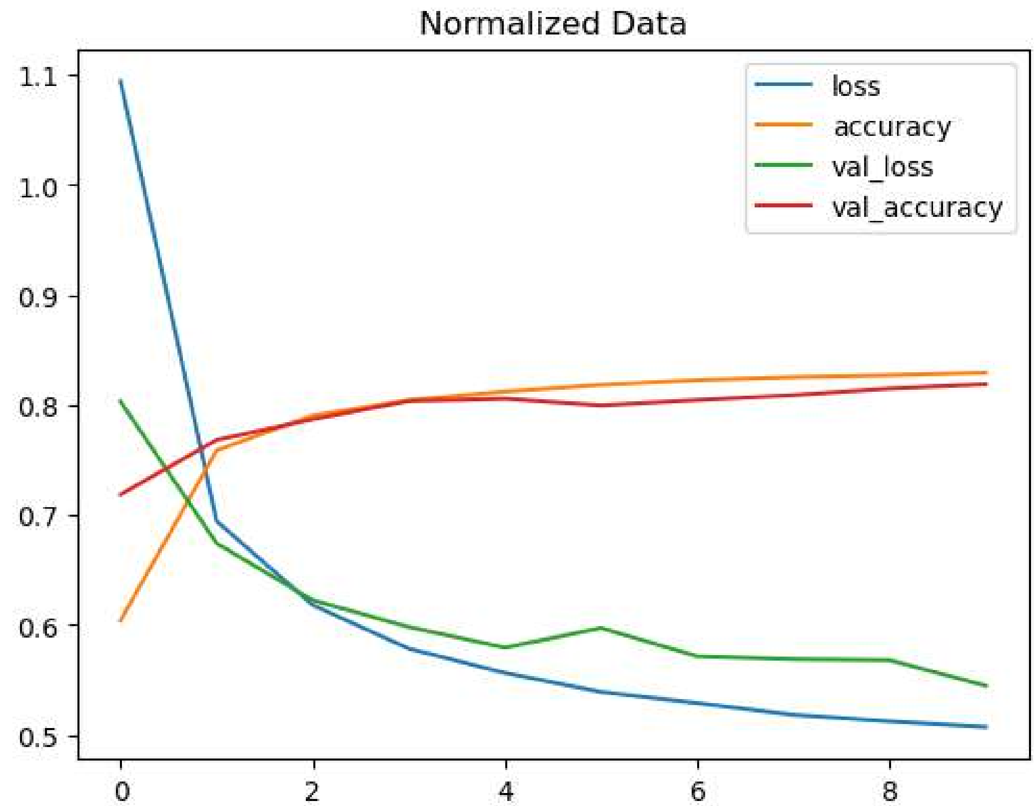
Out[17]:

```
[0.5449552536010742, 0.8188999891281128]
```

In [18]:

```
import pandas as pd

pd.DataFrame(non_norm_history.history).plot(title = "Non_Normalized Data")
pd.DataFrame(norm_history.history).plot(title = "Normalized Data");
```





In [19]:

```
# Set random seed
tf.random.set_seed(42)

# Create the model
```

```
model_3 = tf.keras.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)), # input layer (we had to
reshape 28x28 to 784)
  tf.keras.layers.Dense(4, activation="relu"),
  tf.keras.layers.Dense(4, activation="relu"),
  tf.keras.layers.Dense(10, activation="softmax") # output shape is 10,
activation is softmax
])

# Compile the model
model_3.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(),
                 optimizer=tf.keras.optimizers.Adam(),
                 metrics=["accuracy"])

# Create the learning rate callback
lr_scheduler = tf.keras.callbacks.LearningRateScheduler(lambda epoch: 1e-3
* 10**(epoch/20))

# Fit the model
find_lr_history = model_3.fit(train_data,
                               train_labels,
                               epochs=40, # model already doing pretty good
with current LR, probably don't need 100 epochs
                               validation_data=(test_data, test_labels),
                               callbacks=[lr_scheduler])
```

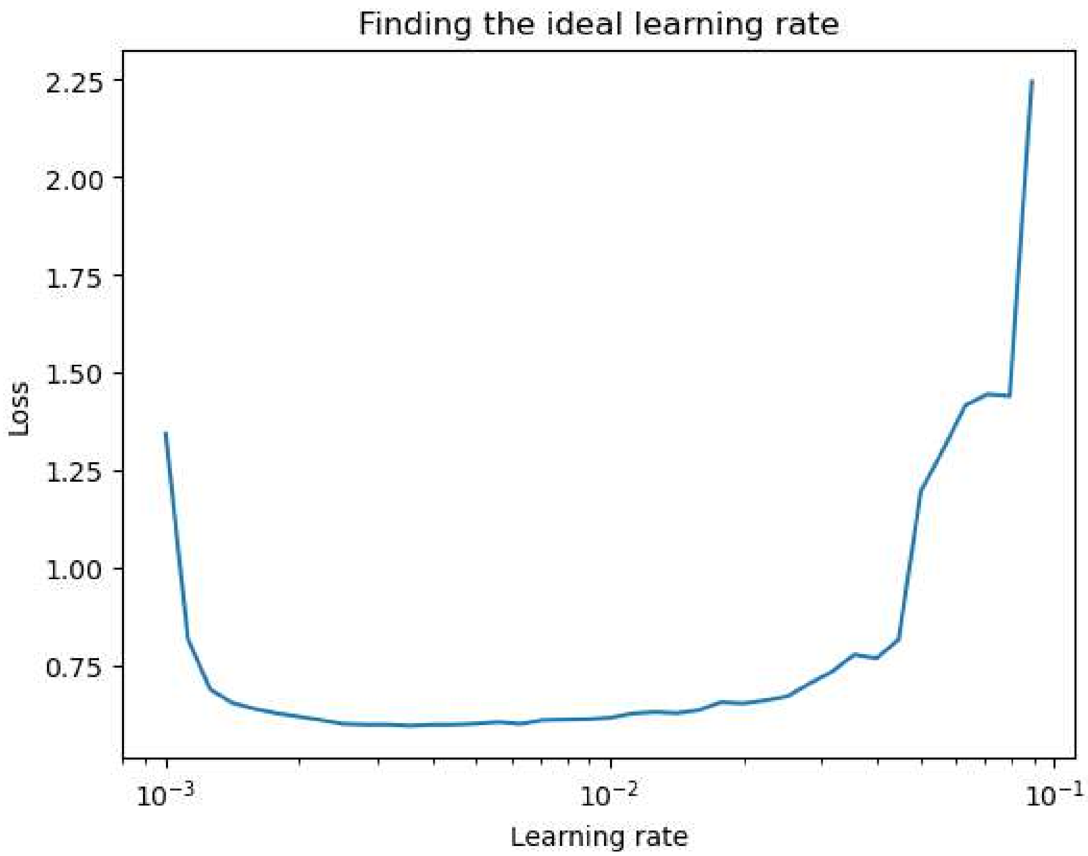
```
Epoch 1/40
1875/1875 [==============================] - 5s 2ms/step - loss: 1.3438 - a
ccuracy: 0.5106 - val_loss: 0.9497 - val_accuracy: 0.6070 - lr: 0.0010
Epoch 2/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.8170 - a
ccuracy: 0.6960 - val_loss: 0.7364 - val_accuracy: 0.7204 - lr: 0.0011
Epoch 3/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6895 - a
ccuracy: 0.7342 - val_loss: 0.6869 - val_accuracy: 0.7349 - lr: 0.0013
Epoch 4/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6542 - a
ccuracy: 0.7522 - val_loss: 0.6793 - val_accuracy: 0.7453 - lr: 0.0014
Epoch 5/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6394 - a
ccuracy: 0.7622 - val_loss: 0.6550 - val_accuracy: 0.7621 - lr: 0.0016
Epoch 6/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6277 - a
ccuracy: 0.7741 - val_loss: 0.6594 - val_accuracy: 0.7573 - lr: 0.0018
Epoch 7/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6193 - a
ccuracy: 0.7761 - val_loss: 0.6478 - val_accuracy: 0.7709 - lr: 0.0020
Epoch 8/40
1875/1875 [==============================] - 5s 2ms/step - loss: 0.6104 - a
ccuracy: 0.7807 - val_loss: 0.6330 - val_accuracy: 0.7771 - lr: 0.0022
Epoch 9/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6006 - a
ccuracy: 0.7839 - val_loss: 0.6472 - val_accuracy: 0.7752 - lr: 0.0025
Epoch 10/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5986 - a
ccuracy: 0.7843 - val_loss: 0.6316 - val_accuracy: 0.7797 - lr: 0.0028
Epoch 11/40
```

```
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5987 - a
ccuracy: 0.7853 - val_loss: 0.6286 - val_accuracy: 0.7770 - lr: 0.0032
Epoch 12/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5959 - a
ccuracy: 0.7871 - val_loss: 0.6260 - val_accuracy: 0.7811 - lr: 0.0035
Epoch 13/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5984 - a
ccuracy: 0.7857 - val_loss: 0.6550 - val_accuracy: 0.7693 - lr: 0.0040
Epoch 14/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5986 - a
ccuracy: 0.7862 - val_loss: 0.6258 - val_accuracy: 0.7813 - lr: 0.0045
Epoch 15/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6012 - a
ccuracy: 0.7828 - val_loss: 0.6284 - val_accuracy: 0.7820 - lr: 0.0050
Epoch 16/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6052 - a
ccuracy: 0.7817 - val_loss: 0.6252 - val_accuracy: 0.7760 - lr: 0.0056
Epoch 17/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6005 - a
ccuracy: 0.7838 - val_loss: 0.6538 - val_accuracy: 0.7667 - lr: 0.0063
Epoch 18/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6105 - a
ccuracy: 0.7792 - val_loss: 0.6306 - val_accuracy: 0.7754 - lr: 0.0071
Epoch 19/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6116 - a
ccuracy: 0.7772 - val_loss: 0.7623 - val_accuracy: 0.7386 - lr: 0.0079
Epoch 20/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6126 - a
ccuracy: 0.7774 - val_loss: 0.6586 - val_accuracy: 0.7722 - lr: 0.0089
Epoch 21/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6162 - a
ccuracy: 0.7751 - val_loss: 0.6846 - val_accuracy: 0.7552 - lr: 0.0100
Epoch 22/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6270 - a
ccuracy: 0.7728 - val_loss: 0.7211 - val_accuracy: 0.7360 - lr: 0.0112
Epoch 23/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6311 - a
ccuracy: 0.7735 - val_loss: 0.7088 - val_accuracy: 0.7585 - lr: 0.0126
Epoch 24/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6281 - a
ccuracy: 0.7773 - val_loss: 0.6276 - val_accuracy: 0.7815 - lr: 0.0141
Epoch 25/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6360 - a
ccuracy: 0.7756 - val_loss: 0.6917 - val_accuracy: 0.7722 - lr: 0.0158
Epoch 26/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6560 - a
ccuracy: 0.7685 - val_loss: 0.6961 - val_accuracy: 0.7545 - lr: 0.0178
Epoch 27/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6531 - a
ccuracy: 0.7710 - val_loss: 0.6983 - val_accuracy: 0.7722 - lr: 0.0200
Epoch 28/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6606 - a
ccuracy: 0.7717 - val_loss: 0.6569 - val_accuracy: 0.7839 - lr: 0.0224
Epoch 29/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6712 - a
ccuracy: 0.7672 - val_loss: 0.7255 - val_accuracy: 0.7614 - lr: 0.0251
Epoch 30/40
```

```
1875/1875 [==============================] - 4s 2ms/step - loss: 0.7045 - a
ccuracy: 0.7510 - val_loss: 0.7446 - val_accuracy: 0.7340 - lr: 0.0282
Epoch 31/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.7344 - a
ccuracy: 0.7416 - val_loss: 0.7522 - val_accuracy: 0.7201 - lr: 0.0316
Epoch 32/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.7780 - a
ccuracy: 0.7266 - val_loss: 0.7452 - val_accuracy: 0.7415 - lr: 0.0355
Epoch 33/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.7687 - a
ccuracy: 0.7319 - val_loss: 0.8045 - val_accuracy: 0.7363 - lr: 0.0398
Epoch 34/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.8161 - a
ccuracy: 0.7195 - val_loss: 1.3783 - val_accuracy: 0.5722 - lr: 0.0447
Epoch 35/40
1875/1875 [==============================] - 4s 2ms/step - loss: 1.1965 - a
ccuracy: 0.5313 - val_loss: 1.2926 - val_accuracy: 0.4457 - lr: 0.0501
Epoch 36/40
1875/1875 [==============================] - 5s 2ms/step - loss: 1.3019 - a
ccuracy: 0.4493 - val_loss: 1.3733 - val_accuracy: 0.4161 - lr: 0.0562
Epoch 37/40
1875/1875 [==============================] - 5s 3ms/step - loss: 1.4159 - a
ccuracy: 0.3964 - val_loss: 1.4079 - val_accuracy: 0.3988 - lr: 0.0631
Epoch 38/40
1875/1875 [==============================] - 5s 2ms/step - loss: 1.4445 - a
ccuracy: 0.3884 - val_loss: 1.3771 - val_accuracy: 0.4137 - lr: 0.0708
Epoch 39/40
1875/1875 [==============================] - 4s 2ms/step - loss: 1.4401 - a
ccuracy: 0.3904 - val_loss: 2.1525 - val_accuracy: 0.1883 - lr: 0.0794
Epoch 40/40
1875/1875 [==============================] - 5s 3ms/step - loss: 2.2454 - a
ccuracy: 0.1330 - val_loss: 2.3139 - val_accuracy: 0.1000 - lr: 0.0891
```

In [20]:

```python
# Plot the learning rate decay curve
import numpy as np
import matplotlib.pyplot as plt
lrs = 1e-3 * (10**(np.arange(40)/20))
plt.semilogx(lrs, find_lr_history.history["loss"]) # want the x-axis to be
log-scale
plt.xlabel("Learning rate")
plt.ylabel("Loss")
plt.title("Finding the ideal learning rate");
```



In [21]:

```
# Set random seed
tf.random.set_seed(42)

# Create the model
model_4 = tf.keras.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)), # input layer (we had to
reshape 28x28 to 784)
  tf.keras.layers.Dense(4, activation="relu"),
  tf.keras.layers.Dense(4, activation="relu"),
  tf.keras.layers.Dense(10, activation="softmax") # output shape is 10,
activation is softmax
])

# Compile the model
model_4.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(),
                 optimizer=tf.keras.optimizers.Adam(lr=0.001), # ideal
learning rate (same as default)
                 metrics=["accuracy"])

# Fit the model
history = model_4.fit(train_data,
                       train_labels,
                       epochs=20,
                       validation_data=(test_data, test_labels))
```

```
WARNING:absl:`lr` is deprecated, please use `learning_rate` instead, or use
the legacy optimizer, e.g.,tf.keras.optimizers.legacy.Adam.
Epoch 1/20
```

```
1875/1875 [==============================] - 6s 3ms/step - loss: 1.3086 - a
ccuracy: 0.5312 - val_loss: 0.8392 - val_accuracy: 0.7246
Epoch 2/20
1875/1875 [==============================] - 4s 2ms/step - loss: 0.7425 - a
ccuracy: 0.7577 - val_loss: 0.7183 - val_accuracy: 0.7587
Epoch 3/20
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6570 - a
ccuracy: 0.7760 - val_loss: 0.6678 - val_accuracy: 0.7707
Epoch 4/20
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6142 - a
ccuracy: 0.7858 - val_loss: 0.6317 - val_accuracy: 0.7805
Epoch 5/20
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5916 - a
ccuracy: 0.7913 - val_loss: 0.6216 - val_accuracy: 0.7837
Epoch 6/20
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5772 - a
ccuracy: 0.7946 - val_loss: 0.6239 - val_accuracy: 0.7811
Epoch 7/20
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5678 - a
ccuracy: 0.7953 - val_loss: 0.6369 - val_accuracy: 0.7670
Epoch 8/20
1875/1875 [==============================] - 5s 2ms/step - loss: 0.5600 - a
ccuracy: 0.7975 - val_loss: 0.5916 - val_accuracy: 0.7913
Epoch 9/20
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5525 - a
ccuracy: 0.8015 - val_loss: 0.5889 - val_accuracy: 0.7898
Epoch 10/20
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5483 - a
ccuracy: 0.8029 - val_loss: 0.5812 - val_accuracy: 0.7961
Epoch 11/20
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5451 - a
ccuracy: 0.8030 - val_loss: 0.5850 - val_accuracy: 0.7955
Epoch 12/20
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5409 - a
ccuracy: 0.8053 - val_loss: 0.5947 - val_accuracy: 0.7879
Epoch 13/20
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5400 - a
ccuracy: 0.8058 - val_loss: 0.5816 - val_accuracy: 0.7948
Epoch 14/20
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5372 - a
ccuracy: 0.8068 - val_loss: 0.5916 - val_accuracy: 0.7951
Epoch 15/20
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5360 - a
ccuracy: 0.8068 - val_loss: 0.5869 - val_accuracy: 0.7955
Epoch 16/20
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5322 - a
ccuracy: 0.8086 - val_loss: 0.5875 - val_accuracy: 0.7959
Epoch 17/20
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5297 - a
ccuracy: 0.8109 - val_loss: 0.5790 - val_accuracy: 0.7983
Epoch 18/20
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5282 - a
ccuracy: 0.8097 - val_loss: 0.5782 - val_accuracy: 0.7975
Epoch 19/20
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5272 - a
ccuracy: 0.8096 - val_loss: 0.5779 - val_accuracy: 0.7991
Epoch 20/20
```

```
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5248 - a
ccuracy: 0.8116 - val_loss: 0.5880 - val_accuracy: 0.7982
```

In [22]:

```
model_4.evaluate(test_data , test_labels)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.5880 - acc
uracy: 0.7982
```

Out[22]:

```
[0.5880288481712341, 0.7982000112533569]
```

In [23]:

```
# Note: The following confusion matrix code is a remix of Scikit-Learn's
# plot_confusion_matrix function - https://scikit-
learn.org/stable/modules/generated/sklearn.metrics.plot_confusion_matrix.ht
ml
# and Made with ML's introductory notebook -
https://github.com/GokuMohandas/MadeWithML/blob/main/notebooks/08_Neural_Ne
tworks.ipynb
import itertools
from sklearn.metrics import confusion_matrix

# Our function needs a different name to sklearn's plot_confusion_matrix
def make_confusion_matrix(y_true, y_pred, classes=None, figsize=(10, 10),
text_size=15):
  """Makes a labelled confusion matrix comparing predictions and ground
truth labels.

  If classes is passed, confusion matrix will be labelled, if not, integer
class values
  will be used.

  Args:
    y_true: Array of truth labels (must be same shape as y_pred).
    y_pred: Array of predicted labels (must be same shape as y_true).
    classes: Array of class labels (e.g. string form). If `None`, integer
labels are used.
    figsize: Size of output figure (default=(10, 10)).
    text_size: Size of output figure text (default=15).

  Returns:
    A labelled confusion matrix plot comparing y_true and y_pred.

  Example usage:
    make_confusion_matrix(y_true=test_labels, # ground truth test labels
                          y_pred=y_preds, # predicted labels
                          classes=class_names, # array of class label names
                          figsize=(15, 15),
                          text_size=10)
  """
  # Create the confustion matrix
  cm = confusion_matrix(y_true, y_pred)
  cm_norm = cm.astype("float") / cm.sum(axis=1)[:, np.newaxis] # normalize
it
  n_classes = cm.shape[0] # find the number of classes we're dealing with

  # Plot the figure and make it pretty
  fig, ax = plt.subplots(figsize=figsize)
```

```
  cax = ax.matshow(cm, cmap=plt.cm.Blues) # colors will represent how
'correct' a class is, darker == better
  fig.colorbar(cax)

  # Are there a list of classes?
  if classes:
    labels = classes
  else:
    labels = np.arange(cm.shape[0])

  # Label the axes
  ax.set(title="Confusion Matrix",
         xlabel="Predicted label",
         ylabel="True label",
         xticks=np.arange(n_classes), # create enough axis slots for each
class
         yticks=np.arange(n_classes),
         xticklabels=labels, # axes will labeled with class names (if they
exist) or ints
         yticklabels=labels)

  # Make x-axis labels appear on bottom
  ax.xaxis.set_label_position("bottom")
  ax.xaxis.tick_bottom()

  # Set the threshold for different colors
  threshold = (cm.max() + cm.min()) / 2.

  # Plot the text on each cell
  for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, f"{cm[i, j]} ({cm_norm[i, j]*100:.1f}%)",
             horizontalalignment="center",
             color="white" if cm[i, j] > threshold else "black",
             size=text_size)
```

In [24]:

```
# Make predictions with the most recent model
y_probs = model_4.predict(test_data) # "probs" is short for probabilities

# View the first 5 predictions
y_probs[:5]
```

```
313/313 [==============================] - 1s 1ms/step
```

Out[24]:

```
array([[1.8819177e-05, 5.9514539e-05, 5.7718548e-06, 1.0002871e-03,
        1.5290198e-05, 6.8496801e-02, 8.8263623e-06, 1.1130547e-01,
        1.5980313e-03, 8.1749123e-01],
       [5.4111497e-06, 1.0108788e-09, 9.0220422e-01, 4.4720648e-05,
        8.1176661e-02, 3.3231482e-27, 1.6238034e-02, 0.0000000e+00,
        3.3093410e-04, 1.7893693e-35],
       [1.8032774e-04, 9.9550748e-01, 1.6358656e-07, 4.3112091e-03,
        4.7549822e-08, 2.0096446e-07, 5.8826981e-07, 3.8959014e-26,
        2.1009447e-10, 1.5644499e-13],
       [2.0412088e-05, 9.9666566e-01, 3.1088121e-08, 3.3022126e-03,
        1.6353068e-08, 1.1563722e-05, 5.9721280e-08, 1.3083310e-20,
        9.8489195e-10, 1.1023541e-08],
       [1.7771378e-01, 6.2479242e-04, 2.1038692e-01, 4.9746882e-02,
        4.4612084e-02, 8.3879383e-09, 5.1447946e-01, 4.8030271e-20,
```

```
        2.4360507e-03, 3.5214108e-16]], dtype=float32)
```

In [25]:

```
y_preds = y_probs.argmax(axis=1)
```

In [26]:

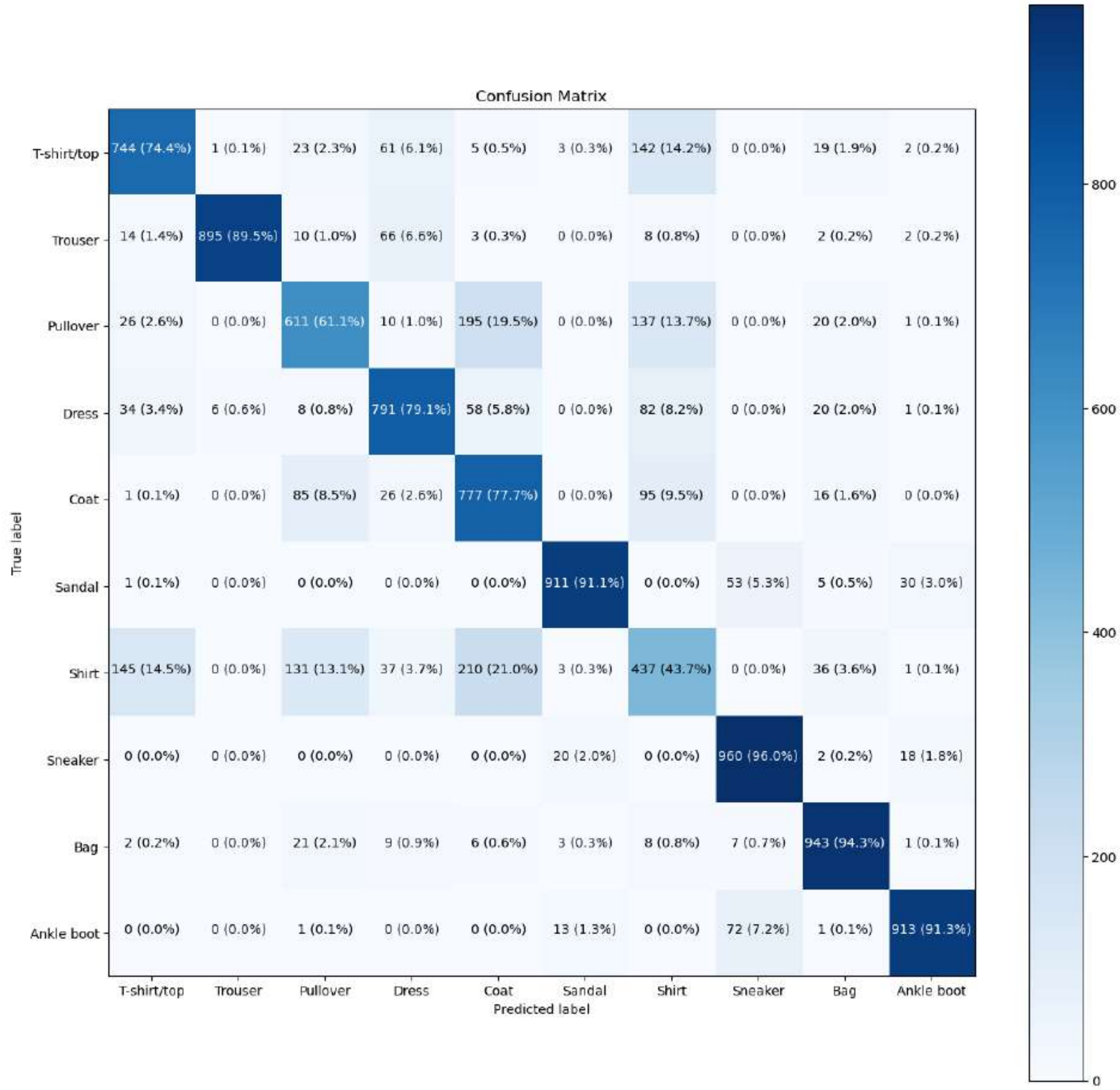
```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_true=test_labels,
                 y_pred=y_preds)
```

Out[26]:

```
array([[744,   1,  23,  61,   5,   3, 142,   0,  19,   2],
       [ 14, 895,  10,  66,   3,   0,   8,   0,   2,   2],
       [ 26,   0, 611,  10, 195,   0, 137,   0,  20,   1],
       [ 34,   6,   8, 791,  58,   0,  82,   0,  20,   1],
       [  1,   0,  85,  26, 777,   0,  95,   0,  16,   0],
       [  1,   0,   0,   0,   0, 911,   0,  53,   5,  30],
       [145,   0, 131,  37, 210,   3, 437,   0,  36,   1],
       [  0,   0,   0,   0,   0,  20,   0, 960,   2,  18],
       [  2,   0,  21,   9,   6,   3,   8,   7, 943,   1],
       [  0,   0,   1,   0,   0,  13,   0,  72,   1, 913]], dtype=int64)
```

In [27]:

```
# Make a prettier confusion matrix
make_confusion_matrix(y_true=test_labels,
                      y_pred=y_preds,
                      classes=class_names,
                      figsize=(15, 15),
                      text_size=10)
```



In [31]:

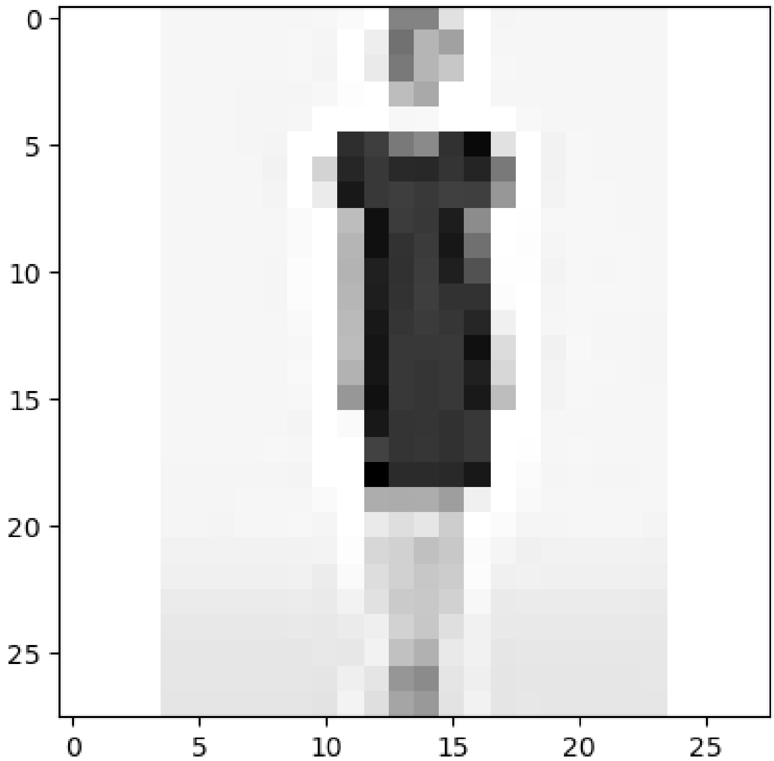
```
import random

# Create a function for plotting a random image along with its prediction
def plot_random_image(model, images, true_labels, classes):
  """Picks a random image, plots it and labels it with a predicted and
truth label.

  Args:
    model: a trained model (trained on data similar to what's in images).
    images: a set of random images (in tensor form).
    true_labels: array of ground truth labels for images.
    classes: array of class names for images.

  Returns:
    A plot of a random image from `images` with a predicted class label
from `model`
    as well as the truth class label from `true_labels`.
  """
  # Setup random integer
```

```
  i = random.randint(0, len(images))

  # Create predictions and targets
  target_image = images[i]
  pred_probs = model.predict(target_image.reshape(1, 28, 28)) # have to
reshape to get into right size for model
  pred_label = classes[pred_probs.argmax()]
  true_label = classes[true_labels[i]]

  # Plot the target image
  plt.imshow(target_image, cmap=plt.cm.binary)

  # Change the color of the titles depending on if the prediction is right
or wrong
  if pred_label == true_label:
    color = "green"
  else:
    color = "red"
```

In [33]:

```
# Check out a random image as well as its prediction
plot_random_image(model=model_4,
                  images=test_data,
                  true_labels=test_labels,
                  classes=class_names)
```

```
1/1 [==============================] - 0s 33ms/step
```



Conclusion:

Obtained results evidence that classifying fashion products with CNN can be more accurate than by using other conventional machine learning models. In addition, it was observed that the dropout technique together with more convolutive layers are effective when it comes to reducing the bias of a model.

Using TensorFlow 2 and GPU for training, we could reach not only a better training time, but also, better accuracies. Table 2 shows the differences between our original work and the present.

REFERENCES:

[1] Hu, W., Huang, Y., Wei, L., Zhang, F., & Li, H. (2015). Deep convolutional neural networks for hyper spectral image classification. Journal of Sensors, 2015.

[2] Krizhevsky, A., Sutskever, L., & Hinton, G. E. (2012). Image Net classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

[3] https://www.researchgate.net/publication/340237897_Classification_of_Garments_from_Fas