Gurjus Singh

MSDS 458 Artificial Intelligence and Deep Learning

January 24th, 2021

Week 3: A.1 First Research/Programming Assignment MNIST Classification

Abstract

In this research I explored the MNIST dataset. This dataset contains handwritten number images. The goal of this research was to explore using multiclass classification using a neural network and supervised learning. I wanted to use some metrics to see how the Neural Network was classifying the images. I also performed dimensionality reduction using PCA and random forests to see if the neural network could still classify with great results.

Introduction

The purpose of this research was to use the MNIST dataset of handwritten digits 0-9 to do supervised learning multiclass classification using Neural Networks. The Neural Network architecture and complexity was explored as well as the activation values in the research. Five experiments were done, and one neural network experiment was which had the best results was chosen for the management recommendation. I also explored lowering the dimensions of the dataset using PCA, and Random Forests and wanted to see if the dataset had same or better results.

Literature Review

Much research has been done with the MNIST dataset. The dataset was originally introduced by LeCun et al. in 1998 (Baldominos et al., 2019, pg. 1). Lecun has also published research of models that have been used on the MNIST dataset (Baldominos et al., 2019, pg. 3). Models include Linear Classifiers which error rate for classification of MNIST dataset is 7.6

percent to 12 percent. Other models include K Nearest-Neighbor which has an error rate of 1.1 percent to 5 percent, nonlinear classifiers which have 5 percent error, Neural Networks have 1.6 percent to 4.7 percent and Convolution Neural Networks which have the lowest error rate at 0.7 percent to 1.7 percent (Baldominos et al., 2019, pg. 3). Other researchers that have worked with MNIST, have also expanded the dataset to include 50,000 more test images, fearful that their models were overfitting (Synced, 2019, para 2).

Methods

My research was first started by importing the packages in 1-1. The Most important packages to import was the numpy packages which was to manipulate arrays, tensor flow package which was used to use key metrics, and Keras package which was used to create the Deep Neural Network architecture, train the model. After the import statements, I made sure the Tensorflow package was 2.0 or above. I then used the **load_dataset** function in Keras to load the MNIST dataset which was then split into **Train and Test** after loading. I then looked at the shapes of the Train and Test dataset which were **(60,000,784) and (10,000, 784).** I also observed the first 10 target labels of the dataset, which gave me a sense of what numbers were in the dataset. I then tried to determine most common numbers in the Train and Test datasets.

1-1 Import Packages

```
# Helper libraries
import datetime
from packaging import version
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.ensemble import RandomForestClassifier

from collections import Counter
import numpy as np
import pandas as pd

# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow.keras.utils import to_categorical
from tensorflow import keras
from tensorflow.keras import models
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
#from plot_keras_history import plot_history
```

```
fig = plt.figure(figsize = (15, 9))

for i in range(50):
    plt.subplot(5, 10, 1+i)
    plt.title(y_train[i])
    plt.xticks([])
    plt.yticks([])
    plt.imshow(x_train[i].reshape(28,28), cmap='binary')
```

1-2 Numbers in MNIST

After seeing the most common number I used the matplotlib package to see the images of the numbers seen by the code and pictures in 1-2. The function imshow() in python helped me to show the images. From the initial observation I saw a 9 that looked like a G and a 5 that looked like a 6. After looking at the initial data, the next step was to do one hot encoding. This was used to convert each data target label into an array of 10 elements where the target value was represented by index and filled with an one if it matched, and 0s elsewhere. This was done using **to_categorical** method. After one hot encoding it was time to normalize the MNIST dataset. This was done by dividing the pixel values by 255. This was to get the values between 0 and 1.

After the data preparation, data visualization, it was time to create the models. I did a total of 10 experiments. The first two models had 128 1st layer nodes, and 1 and 2 hidden layers respectively. On the third layer I used a total of 5 hidden layers. I used this architecture for experiments for 4 and 5, but what was interesting about experiments 4 and 5 was that I used PCA, a dimensionality reduction algorithm, which reduced the features from 784 to 154. I then reduced the features even further with Random Forests to 70-pixel columns and wanted to see if the accuracy, and results were great as well. For Experiments 5-10, I wanted to see what

increasing the number of nodes would do in each of the two layers, so I increased to 64 in Experiment 6 to 512 in Experiment 10 in both layers.

For compiling the Deep Neural Network models, the loss function I used was categorical cross entropy. A loss functions helps the neural network to find the right predictions. It uses a distance score to train the neural network (Chollet, 2017, pg. 10). The optimizer which I used was rmsprop. The optimizer is used in Back Propagation to adjust the weights so the loss function can lower its distance between predictions and target values (Chollet, 2017, pg. 11). The way back propagation works is by applying the "chain rule" to different parameters and in order to find out how much each parameter in the neural network contributed to the final answer (Chollet, 2017, pg. 52).

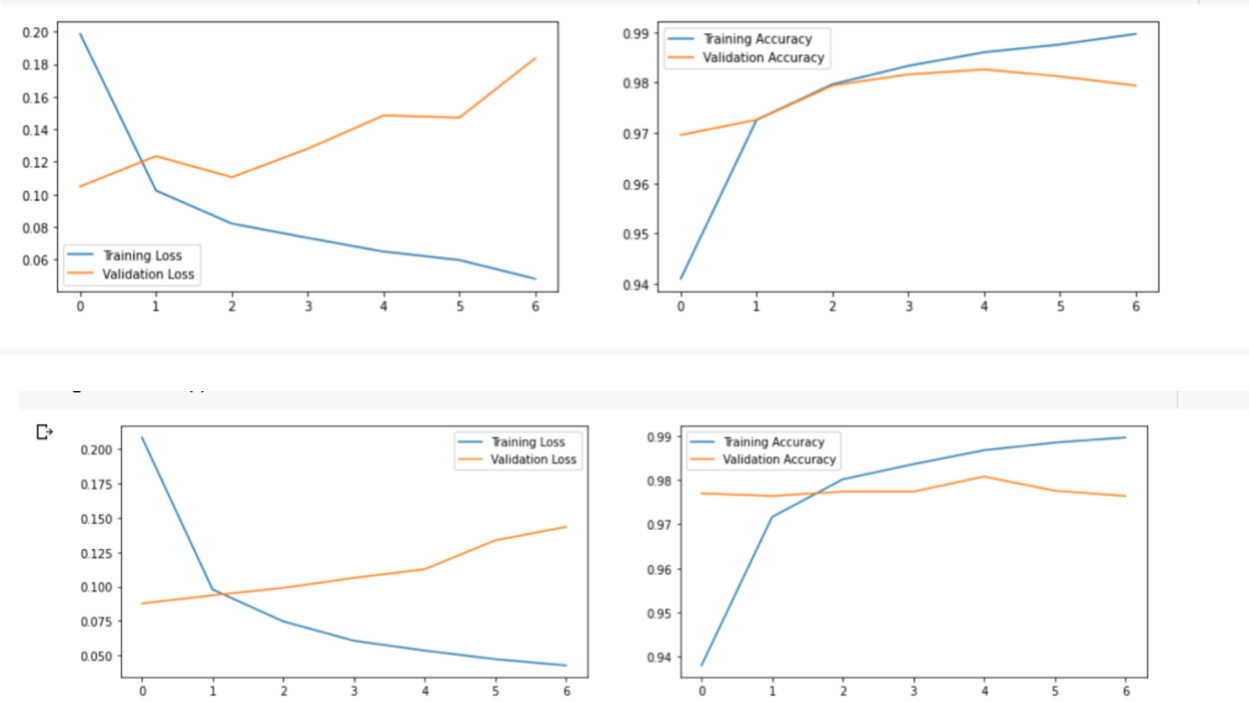
Results

Experiments	Layers Used	Nodes per Layer	Train Accuracy	Val Accuracy	Test Accuracy
1	2	128 - 1st layer; 1 - hidden layer	54.03%	53.52%	53.92%
2	2	128 - 1st layer; 2 - hidden layer	11.36%	10.58%	11.35%
3	2	128 - 1st layer; 5 - hidden layer	98.57%	97.12%	96.82%
4	2	PCA 154; 128 - 1st layer; 5 - hidden laye	98.33%	96.07%	96.60%
5	2	Random Forest 70; 128 nodes - 1st layer; 5 nodes - hidden layer	92.77%	93.52%	91.92%
6	2	64 Nodes	98.51%	97.62%	97.35%
7	2	128 Nodes	99.18%	97.96%	97.79%
8	2	128 - 1st layer; 256 - hidden layer	99.10%	97.92%	97.90%
9	2	256 Nodes	98.93%	97.60%	97.43%
10	2	512 Nodes	99.22%	97.36%	97.54%

Experiments figure 1-3

I created a series of ten experiments in using hidden nodes from 1-512, and 1st layer number of nodes from 128-512. I wanted to find out what number of nodes got the accuracy just about right? I compared my results and most of the time my models were overfitting. Particularly, experiments 1 and 2 were underfitting as seen in 1-3 which had hidden nodes of one and two, and first level nodes of 128. As I increased the level of neurons for the hidden layer, I got overfitting as seen in experiments 3 and 4 in figure 1-3 with hidden nodes of 5. Experiment 5 did not overfit, but I wanted to see if train and test accuracy could be improved. With experiment

6 I did improve train accuracy to 98.51 percent and test accuracy to 97.35 percent. Experiments 7-10 also started to overfit as I increased number of nodes again in both layers as I saw from the epochs as seen in 1-4.



Examples of Overfitting Epochs from Experiments 7-10 1-4

After comparing my accuracies of my models and the differences between the train and test accuracies I determined that Experiment 6 which had **64 neurons for both layers** was my best model. I chose 6 over experiment 5 because of the higher of accuracies. I then looked at the confusion matrix for my best model and saw that it was in fact getting the majority of predictions right as seen in the diagonals in 1-5, 1-6. I also looked at the plot of the epochs and made sure the model was not overfitting as seen in 1-7. There was a tiny gap, which meant model was doing decent and was not diverging largely like the other models.

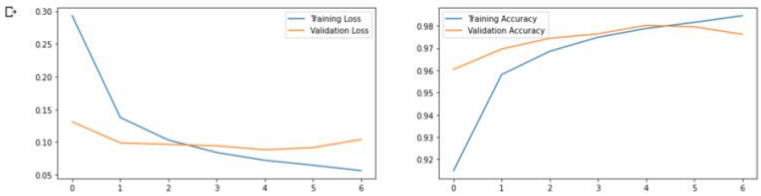
```
<tf.Tensor: shape=(10, 10), dtype=int32, numpy=
array([[5862,    0,    2,    9,    2,    6,   22,    2,   11,    7],
       [   0, 6682,    8,   12,    7,    1,    1,    9,   19,    3],
       [   5,    6, 5850,   50,    7,    1,    1,   22,   13,    3],
       [   0,    3,    9, 6082,    0,   13,    0,   10,    8,    6],
       [   1,    6,    6,    1, 5804,    0,    9,    3,    4,    8],
       [   2,    2,    4,   54,    3, 5308,   12,    0,   20,   16],
       [   7,    8,    3,    1,    3,   10, 5877,    0,    9,    0],
       [   1,    9,    9,    6,    6,    1,    0, 6210,    2,   21],
       [   2,   19,    9,   39,    9,    9,    5,    9, 5726,   24],
       [   3,    4,    1,   28,   82,    6,    1,   26,    9, 5789]], dtype=int32)>
```

Confusion Matrix using Train Set for Exp.6 1-5

```
<tf.Tensor: shape=(10, 10), dtype=int32, numpy=
array([[ 968,    0,    1,    1,    0,    2,    4,    4,    0,    0],
       [   0, 1125,    1,    1,    0,    1,    5,    0,    2,    0],
       [   6,    5, 1001,    1,    3,    0,    4,    4,    8,    0],
       [   1,    0,    7,  971,    2,   17,    0,    5,    5,    2],
       [   1,    1,    5,    0,  967,    0,    2,    0,    1,    5],
       [   2,    1,    0,    2,    1,  876,    7,    1,    1,    1],
       [   4,    3,    0,    1,    6,    6,  938,    0,    0,    0],
       [   1,    6,   10,    3,    0,    0,    0,  992,    3,   13],
       [   7,    1,    6,    4,    7,    9,    9,    4,  921,    6],
       [   3,    3,    1,    6,   22,    9,    0,    4,    2,  959]], dtype=int32)>
```

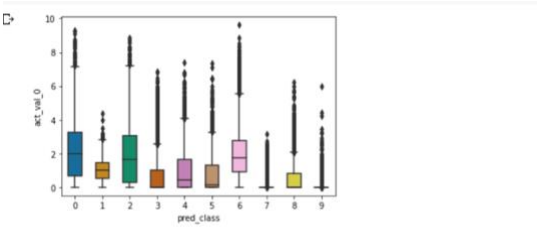
Confusion Matrix using Test Set for Exp.6 1-6

```
plt.figure(figsize=(16, 4))
for i, metrics in enumerate(zip([losses, accs], [val_losses, val_accs], ['Loss', 'Accuracy'])):
    plt.subplot(1, 2, i + 1)
    plt.plot(range(epochs), metrics[0], label='Training {}'.format(metrics[2]))
    plt.plot(range(epochs), metrics[1], label='Validation {}'.format(metrics[2]))
    plt.legend()
plt.show()
```



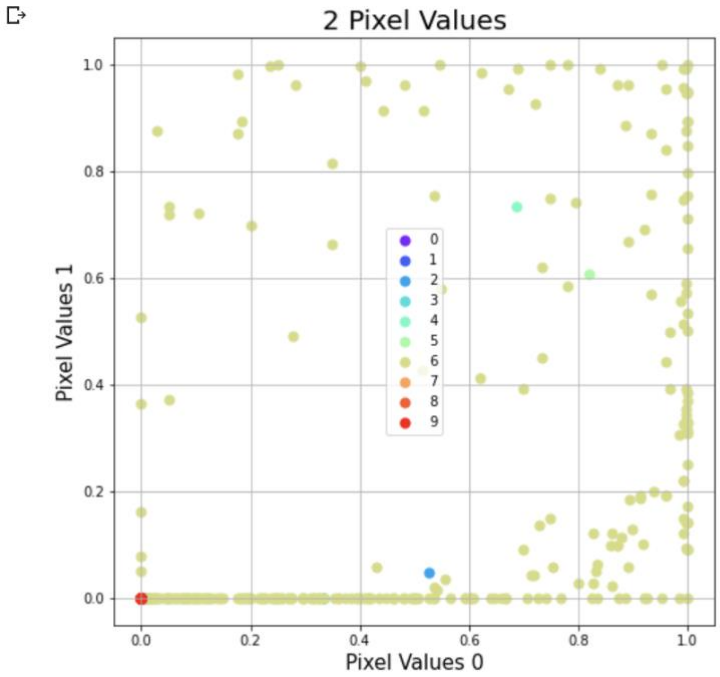
Plot Experiment 6 1-7

I also looked at the 1-8 boxplots which gave me a sense of the range of activation values. I only saw some overlapping between class 1 and 2. Most of the classes had a unique range of activation values and their boxplots were not overlapping which is what is ideally what I wanted.



1-8 boxplots Experiment 6

In 1-9 I then looked at the scatterplot to see how many classes could be predicted through 2 pixels and I found 4-5 classes could be predicted as seen in 1-9.



Experiment 6 2 pixel scatter plot 1-9

Conclusions

In this research, I learned how train deep neural networks on the MNIST dataset. The MNIST dataset as described above, is a set of handwritten number images, which the neural network could recognize. In order to train the model, it required preparing the data into training, validation and test sets. It required normalizing the data from 0 to 1, and it required one hot encoding for target values in the data. This was a supervised learning research method as it used

target variables to train the neural network. After training the models, I evaluated the models, and found most of them were overfitting, and underfitting**.** I found Experiment 6 to be the best which used 64 nodes in 2 layers to fit the data; **therefore, for the management recommendation I recommend the 64 nodes in 2 layers neural network architecture to train using the MNIST data as it had a 98.51 percent training accuracy and a 97.35 percent test accuracy.** I also looked at the activation box plots and small minimum overlap which is ideal as that mean it is distinguishing between each image and each range of activation values in unique. In the future I would like to see how I can get the neural network to predict 100 percent accuracy, while not overfitting.

References

Baldaminos, A., Saez, Y., & Isasi, P. (n.d.). A survey of handwritten character recognition with MNIST and EMNIST. *Applied Sciences*.

Chollet, F. (2017). *Deep Learning with Python*. Manning Publications Company.

Synced. (2019, June 19). MNIST reborn, restored and expanded: Additional 50K training samples. Retrieved January 20, 2021, from https://syncedreview.com/2019/06/19/mnist-reborn-restored-and-expanded-additional-50k-training-samples/

Northwestern

Appendix

MSDS458 Research Assignment 1:

Our `DNN` will consist of 784 input nodes, a hidden layer with `128 nodes` and 10 output nodes (corresponding to the 10 digits). We use `mnist.load_data()` to get the 70,000 images divided into a set of 60,000 training images and 10,000 test images. We hold back 5,000 of the 60,000 training images for validation. After training the model, we group the 60,000 `activation values` of the hidden node for the (original) set of training images by the 10 predicted classes and visualize these sets of values using a `boxplot`. We expect the overlap between the range of values in the "boxes" to be minimal.

Import Packages

Since Keras in part of TensorFlow 2.x, we import keras from tensorflow and use tenserflow.keras.xxx to import all other Keras packages. The seed argument produces a deterministic sequence of tensors across multiple calls.

```
# Helper libraries
import datetime
from packaging import version
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.ensemble import RandomForestClassifier

from collections import Counter
import numpy as np
import pandas as pd
```

```
# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow.keras.utils import to_categorical
from tensorflow import keras
from tensorflow.keras import models
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
#from plot_keras_history import plot_history

%matplotlib inline
np.set_printoptions(precision=3, suppress=True)
```

Verify TensorFlow Version and Keras Version

```
print("This notebook requires TensorFlow 2.0 or above")
print("TensorFlow version: ", tf.__version__)
assert version.parse(tf.__version__).release[0] >=2
```

```
This notebook requires TensorFlow 2.0 or above
TensorFlow version:  2.4.0
```

```
print("Keras version: ", keras.__version__)
```

```
Keras version:  2.4.0
```

Mount Google Drive to Colab Enviorment

```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

Loading MNIST Dataset

The MNIST dataset of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples. It comes prepackaged as part of tf.Keras. Use the `tf.keras.datasets.mnist.load_data` to the get these datasets (and the corresponding labels) as Numpy arrays.

```
(x_train, y_train), (x_test, y_test)= tf.keras.datasets.mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
```

- Tuple of Numpy arrays: (x_train, y_train), (x_test, y_test).
- x_train, x_test: uint8 arrays of grayscale image data with shapes (num_samples, 28, 28).
- y_train, y_test: uint8 arrays of digit labels (integers in range 0-9)

EDA Training and Test Datasets

- Imported 60000 examples for training and 10000 examples for test
- Imported 60000 labels for training and 10000 labels for test

```
print('x_train:\t{}'.format(x_train.shape))
print('y_train:\t{}'.format(y_train.shape))
print('x_test:\t\t{}'.format(x_test.shape))
print('y_test:\t\t{}'.format(y_test.shape))
```

```
x_train:        (60000, 28, 28)
y_train:        (60000,)
x_test:         (10000, 28, 28)
y_test:         (10000,)
```

Review labels for training dataset

```
print("First ten labels training dataset:\n {}\n".format(y_train[0:10]))
```

```
First ten labels training dataset:
 [5 0 4 1 9 2 1 3 1 4]
```

Find frequency of each label in the training and test data

```
Counter(y_train).most_common()
```

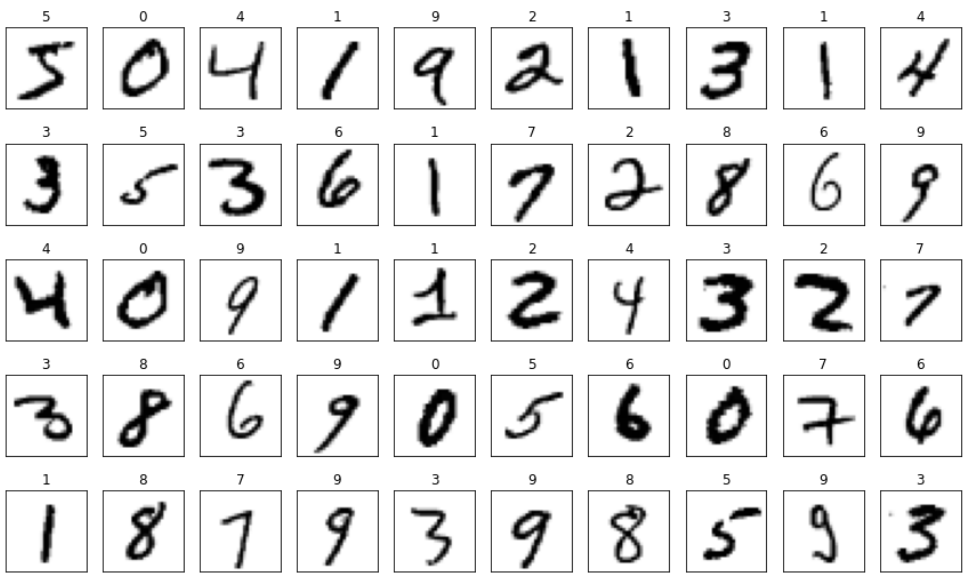
```
[(1, 6742),
 (7, 6265),
 (3, 6131),
 (2, 5958),
 (9, 5949),
 (0, 5923),
 (6, 5918),
 (8, 5851),
 (4, 5842),
 (5, 5421)]
```

```
Counter(y_test).most_common()
```

```
[(1, 1135),
 (2, 1032),
 (7, 1028),
 (3, 1010),
 (9, 1009),
 (4, 982),
 (0, 980),
 (8, 974),
 (6, 958),
 (5, 892)]
```

Plot Examples

```
fig = plt.figure(figsize = (15, 9))

for i in range(50):
    plt.subplot(5, 10, 1+i)
    plt.title(y_train[i])
    plt.xticks([])
    plt.yticks([])
    plt.imshow(x_train[i].reshape(28,28), cmap='binary')
```



Preprocessing Data

One Hot Encoding Labels

We will change the way this label is represented from a class name or number, to a vector of all possible classes with all the classes set to 0 except the one which this example belongs to - which will be set to 1. For example:

original label	one-hot encoded label
5	[0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
7	[0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
1	[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]

[ ] ↳ *2 cells hidden*

Preprocessing the Examples

numpy to unroll the examples from (28, 28) arrays to (784, 1) vectors

example shape before reshape

```
print('x_train:\t{}'.format(x_train.shape))
print('x_test:\t\t{}'.format(x_test.shape))
```

```
x_train:        (60000, 28, 28)
x_test:         (10000, 28, 28)
```

```
x_train_reshaped = np.reshape(x_train, (60000, 784))
x_test_reshaped = np.reshape(x_test, (10000, 784))

print('x_train_reshaped shape: ', x_train_reshaped.shape)
print('x_test_reshaped shape: ', x_test_reshaped.shape)
```

```
x_train_reshaped shape:  (60000, 784)
x_test_reshaped shape:  (10000, 784)
```

1. Each element in each example is a pixel value
2. Pixel values range from 0 to 255
3. 0 = Black
4. 255 = White

```
print(set(x_train_reshaped[0]))
```

```
{0, 1, 2, 3, 9, 11, 14, 16, 18, 23, 24, 25, 26, 27, 30, 35, 36, 39, 43, 45, 46, 49, 55, 56, 64, 66, 70, 78, 80, 81, 82, 90, 93, 94, 107, 108, 114, 119, 126, 127, 130, 132, 133, 135, 136, 139, 148, 150, 154, 156, 160, 166, 170,
```

```
np.set_printoptions(linewidth=np.inf)
print("{}".format(x_train[2020]))
```

```
[[  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   0 167 208  19   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  13 235 254  99   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  74 254 234   4   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 154 254 145   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 224 254  92   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  51 245 211  13   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   2 169 254 101   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0  27 254 254  88   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0  72 255 241  15   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0  88 254 153   0   0   0  33  53 155 156 102  15   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 130 254  31   0 128 235 254 254 254 254 186  10   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 190 254  51 178 254 246 213 111 109 186 254 145   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 192 254 229 254 216  90   0   0   0  57 254 234   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 235 254 254 247  85   0   0   0   0  32 254 234   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 235 254 254 118   0   0   0   0   0 107 254 201   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 235 255 254 102  12   0   0   0   8 188 248 119   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 207 254 254 238 107   0   0  39 175 254 148   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0  84 254 248  74  11  32 115 238 254 176  11   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0  21 214 254 254 254 254 254 254 132   6   0   0   0   0   0   0   0]
```

```
 [  0   0   0   0   0   0   0   0   0   0   0  14  96 176 254 254 214  48  12   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   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]]
```

```
x_train_norm = x_train_reshaped.astype('float32') / 255
x_test_norm = x_test_reshaped.astype('float32') / 255
```

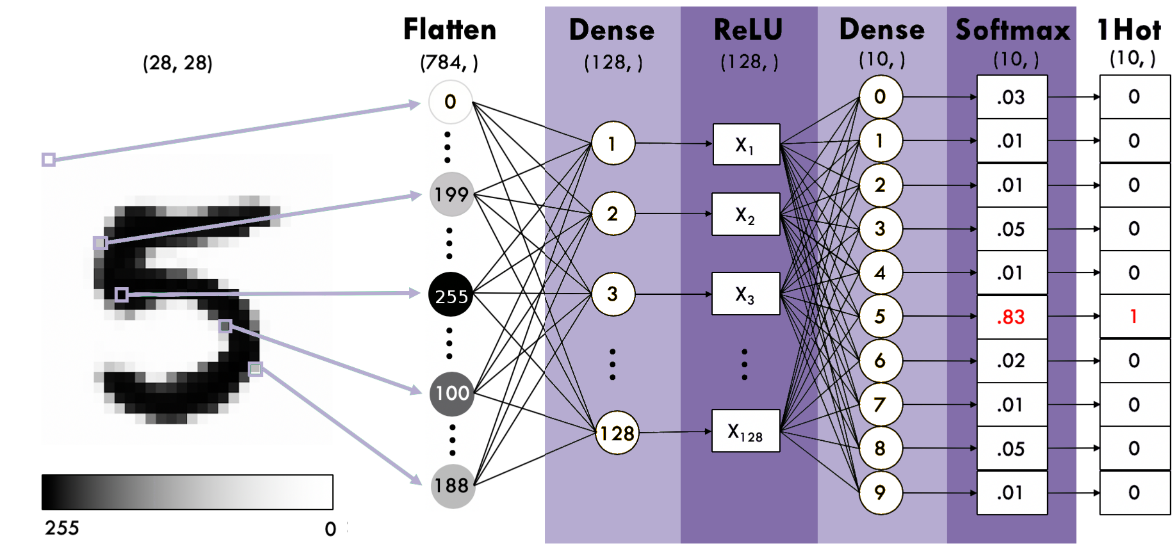
```
print(set(x_train_norm[0]))
```

```
{0.0, 0.011764706, 0.53333336, 0.07058824, 0.49411765, 0.6862745, 0.101960786, 0.6509804, 1.0, 0.96862745, 0.49803922, 0.11764706, 0.14117648, 0.36862746, 0.6039216, 0.6666667, 0.043137256, 0.05490196, 0.03529412, 0.85882354,
```

Create the Model

Specify a network architecture

Below is the neural network architecture we will use today for classifying MNIST digits.



Build DNN Model

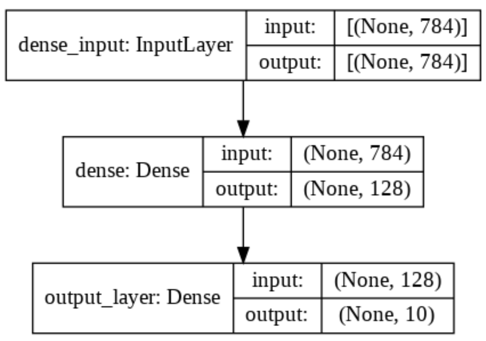
We use a Sequential class defined in Keras to create our model. All the layers are going to be Dense layers. This means, like our examples above, all the nodes of a layer would be connected to all the nodes of the preceding layer i.e. densely connected.

```
model = Sequential([
    Dense(input_shape=[784], units = 128, activation = tf.nn.relu),
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 128)               100480
_________________________________________________________________
output_layer (Dense)         (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

```
keras.utils.plot_model(model, "mnist_model.png", show_shapes=True)
```



Compiling the model

In addition to setting up our model architecture, we also need to define which algorithm should the model use in order to optimize the weights and biases as per the given data. We will use stochastic gradient descent.

We also need to define a loss function. Think of this function as the difference between the predicted outputs and the actual outputs given in the dataset. This loss needs to be minimised in order to have a higher model accuracy. That's what the optimization algorithm essentially does - it minimises the loss during model training. For our multi-class classification problem, categorical cross entropy is commonly used.

Finally, we will use the accuracy during training as a metric to keep track of as the model trains.

tf.keras.optimizers.RMSprop
https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/RMSprop

tf.keras.losses.CategoricalCrossentropy
https://www.tensorflow.org/api_docs/python/tf/keras/losses/CategoricalCrossentropy

```
model.compile(optimizer='rmsprop',
              loss = 'categorical_crossentropy',
              metrics=['accuracy'])
```

Training the model

Attention to the **Callbacks**
ToDo - Early Stopping
Todo - Model Checkpoints

```
history = model.fit(
    x_train_norm
    ,y_train_encoded
    ,epochs = 200
    ,validation_split=0.20
    ,callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

```
Epoch 1/200
```

```
1500/1500 [==============================] - 6s 3ms/step - loss: 0.4440 - accuracy: 0.8758 - val_loss: 0.1744 - val_accuracy: 0.9489
Epoch 2/200
1500/1500 [==============================] - 4s 2ms/step - loss: 0.1441 - accuracy: 0.9575 - val_loss: 0.1185 - val_accuracy: 0.9670
Epoch 3/200
1500/1500 [==============================] - 3s 2ms/step - loss: 0.0958 - accuracy: 0.9724 - val_loss: 0.1131 - val_accuracy: 0.9688
Epoch 4/200
1500/1500 [==============================] - 4s 2ms/step - loss: 0.0734 - accuracy: 0.9787 - val_loss: 0.1090 - val_accuracy: 0.9709
Epoch 5/200
1500/1500 [==============================] - 4s 2ms/step - loss: 0.0621 - accuracy: 0.9826 - val_loss: 0.1038 - val_accuracy: 0.9728
Epoch 6/200
1500/1500 [==============================] - 4s 2ms/step - loss: 0.0504 - accuracy: 0.9857 - val_loss: 0.1095 - val_accuracy: 0.9732
Epoch 7/200
1500/1500 [==============================] - 4s 2ms/step - loss: 0.0426 - accuracy: 0.9884 - val_loss: 0.1039 - val_accuracy: 0.9747
Epoch 8/200
1500/1500 [==============================] - 4s 2ms/step - loss: 0.0356 - accuracy: 0.9896 - val_loss: 0.1041 - val_accuracy: 0.9751
Epoch 9/200
1500/1500 [==============================] - 4s 2ms/step - loss: 0.0334 - accuracy: 0.9912 - val_loss: 0.1135 - val_accuracy: 0.9740
Epoch 10/200
1500/1500 [==============================] - 4s 2ms/step - loss: 0.0275 - accuracy: 0.9930 - val_loss: 0.1216 - val_accuracy: 0.9734
```

validation_data = Data on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data

Test the model

In order to ensure that this is not a simple "memorization" by the machine, we should evaluate the performance on the test set. This is easy to do, we simply use the `evaluate` method on our model.

```
loss, accuracy = model.evaluate(x_test_norm, y_test_encoded)
print('test set accuracy: ', accuracy * 100)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.1097 - accuracy: 0.9727
test set accuracy:  97.26999998092651
```

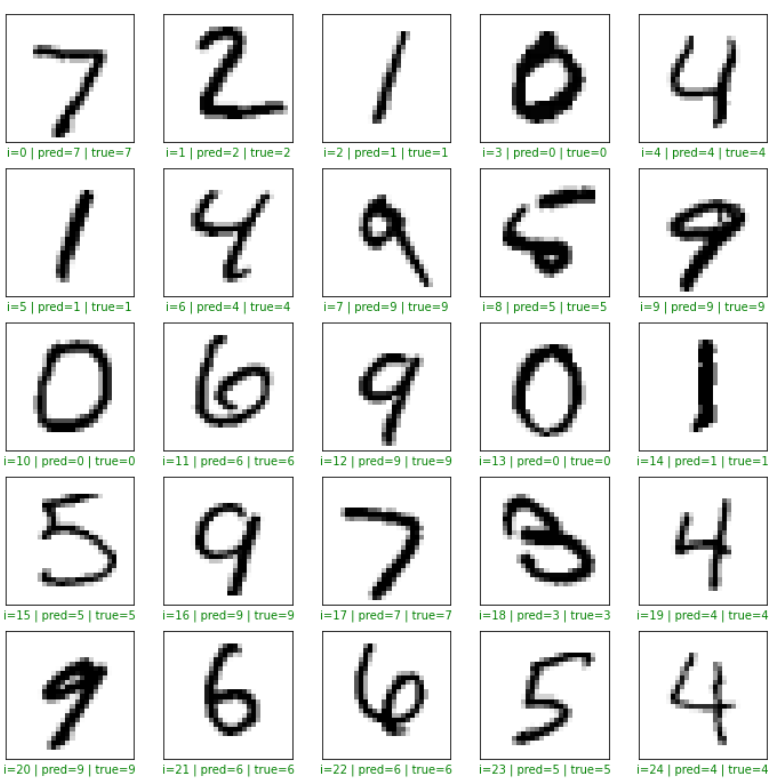
Predictions

```
preds = model.predict(x_test_norm)
print('shape of preds: ', preds.shape)
```

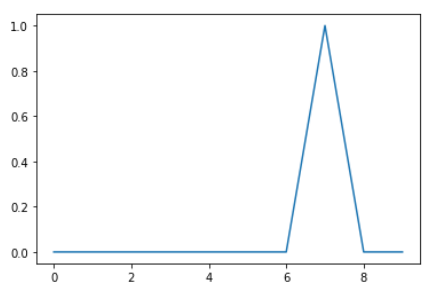
```
shape of preds:  (10000, 10)
```

Look at the first 25 - Plot test set images along with their predicted and actual labels to understand how the trained model actually performed

```
plt.figure(figsize = (12, 12))

start_index = 0

for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    pred = np.argmax(preds[start_index + i])
    actual = np.argmax(y_test_encoded[start_index + i])
    col = 'g'
    if pred != actual:
        col = 'r'
    plt.xlabel('i={} | pred={} | true={}'.format(start_index + i, pred, actual), color = col)
    plt.imshow(x_test[start_index + i], cmap='binary')
plt.show()
```




```
"""
Enter the index value in place of the value 17 below for the prediction
that you want to plot the probability scores for
"""
index = 17

plt.plot(preds[index])
plt.show()
```



Review Performance

```
history_dict = history.history
history_dict.keys()
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

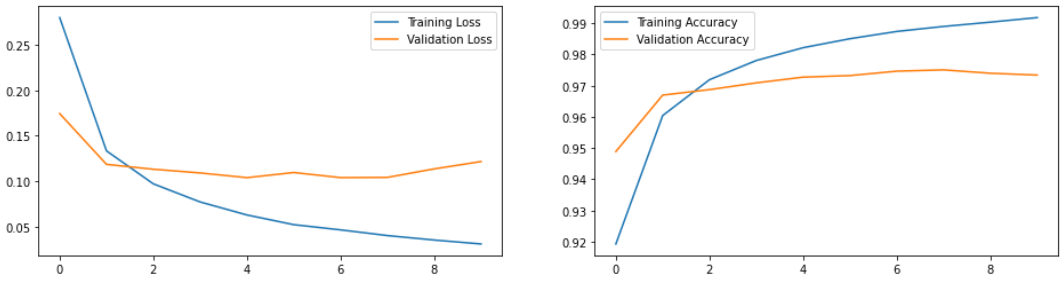
Plotting Performance Metrics

We use Matplotlib to create 2 plots--displaying the training and validation loss (resp. accuracy) for each (training) epoch side by side.

```
#plot_history(history.history)
#plt.show()

history_dict = history.history
history_dict.keys()
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
val_accs = history.history['val_accuracy']
epochs = len(losses)

plt.figure(figsize=(16, 4))
for i, metrics in enumerate(zip([losses, accs], [val_losses, val_accs], ['Loss', 'Accuracy'])):
    plt.subplot(1, 2, i + 1)
    plt.plot(range(epochs), metrics[0], label='Training {}'.format(metrics[2]))
    plt.plot(range(epochs), metrics[1], label='Validation {}'.format(metrics[2]))
    plt.legend()
plt.show()
```



Experiment 1

```
model1 = Sequential([
    Dense(input_shape=[784], units = 128, activation = tf.nn.relu),
    Dense(name = "hidden_layer", units = 1, activation = tf.nn.relu),
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])

model1.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 128)               100480
_________________________________________________________________
hidden_layer (Dense)         (None, 1)                 129
_________________________________________________________________
output_layer (Dense)         (None, 10)                20
=================================================================
Total params: 100,629
Trainable params: 100,629
Non-trainable params: 0
_________________________________________________________________
```

```
model1.compile(optimizer='rmsprop',
               loss = 'categorical_crossentropy',
               metrics=['accuracy'])

history = model1.fit(
    x_train_norm
    ,y_train_encoded
    ,epochs = 200
    ,validation_split=0.0833
    ,callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

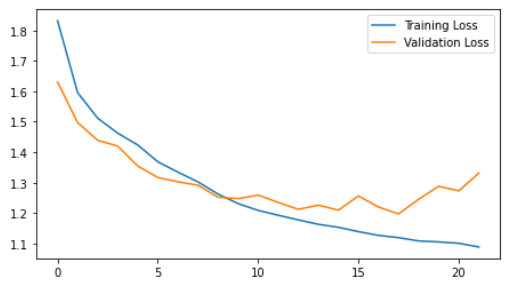
```
Epoch 1/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.9622 - accuracy: 0.2119 - val_loss: 1.6304 - val_accuracy: 0.2975
Epoch 2/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.6252 - accuracy: 0.3221 - val_loss: 1.4971 - val_accuracy: 0.4142
Epoch 3/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.5173 - accuracy: 0.3972 - val_loss: 1.4392 - val_accuracy: 0.4334
Epoch 4/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.4650 - accuracy: 0.4372 - val_loss: 1.4201 - val_accuracy: 0.4368
Epoch 5/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.4335 - accuracy: 0.4662 - val_loss: 1.3545 - val_accuracy: 0.5114
Epoch 6/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.3700 - accuracy: 0.5082 - val_loss: 1.3172 - val_accuracy: 0.5708
Epoch 7/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.3269 - accuracy: 0.5349 - val_loss: 1.3028 - val_accuracy: 0.5714
Epoch 8/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.3022 - accuracy: 0.5640 - val_loss: 1.2917 - val_accuracy: 0.5972
Epoch 9/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.2666 - accuracy: 0.5971 - val_loss: 1.2523 - val_accuracy: 0.6102
Epoch 10/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.2320 - accuracy: 0.6141 - val_loss: 1.2476 - val_accuracy: 0.6016
Epoch 11/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.2092 - accuracy: 0.6150 - val_loss: 1.2594 - val_accuracy: 0.6251
Epoch 12/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.1733 - accuracy: 0.6372 - val_loss: 1.2350 - val_accuracy: 0.6124
Epoch 13/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.1549 - accuracy: 0.6408 - val_loss: 1.2125 - val_accuracy: 0.6541
Epoch 14/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.1482 - accuracy: 0.6488 - val_loss: 1.2263 - val_accuracy: 0.6595
Epoch 15/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.1722 - accuracy: 0.6602 - val_loss: 1.2099 - val_accuracy: 0.6541
Epoch 16/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.1306 - accuracy: 0.6665 - val_loss: 1.2564 - val_accuracy: 0.6613
Epoch 17/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.1197 - accuracy: 0.6656 - val_loss: 1.2200 - val_accuracy: 0.6607
Epoch 18/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.1193 - accuracy: 0.6725 - val_loss: 1.1976 - val_accuracy: 0.6651
Epoch 19/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.1020 - accuracy: 0.6786 - val_loss: 1.2457 - val_accuracy: 0.6703
Epoch 20/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.1015 - accuracy: 0.6809 - val_loss: 1.2881 - val_accuracy: 0.6845
Epoch 21/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.0904 - accuracy: 0.6819 - val_loss: 1.2733 - val_accuracy: 0.6739
Epoch 22/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.0832 - accuracy: 0.6810 - val_loss: 1.3313 - val_accuracy: 0.6805
```

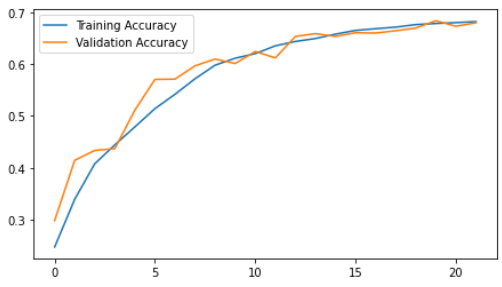
```
loss, accuracy = model1.evaluate(x_test_norm, y_test_encoded)
print('test set accuracy: ', accuracy * 100)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 1.4040 - accuracy: 0.6694
test set accuracy:  66.93999767303467
```

```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
```

```
val_accs = history.history['val_accuracy']
epochs = len(losses)

plt.figure(figsize=(16, 4))
for i, metrics in enumerate(zip([losses, accs], [val_losses, val_accs], ['Loss', 'Accuracy'])):
    plt.subplot(1, 2, i + 1)
    plt.plot(range(epochs), metrics[0], label='Training {}'.format(metrics[2]))
    plt.plot(range(epochs), metrics[1], label='Validation {}'.format(metrics[2]))
    plt.legend()
plt.show()
```






```
pred_classes = np.argmax(model1.predict(x_train_norm), axis=-1)
pred_classes
```

```
array([6, 0, 6, ..., 6, 6, 8])
```

```
conf_mx = tf.math.confusion_matrix(y_train, pred_classes)
conf_mx
```

```
<tf.Tensor: shape=(10, 10), dtype=int32, numpy=
array([[5301,   31,   27,  444,    1,    4,   19,   68,   17,   11],
       [   7, 6518,    9,   29,    0,    1,   26,   62,   73,   17],
       [1010,   21, 4620,  228,    0,    0,   12,   61,    2,    4],
       [ 338,  130,   42, 4907,    0,    1,   21,  634,   42,   16],
       [   3,    8,    0,    8,   60,   69, 5534,    6,   13,  141],
       [   5,   29,    1,   13,   27,   44, 5136,   12,   38,  116],
       [   3,   10,    0,    5,    6,   14, 5817,    4,   18,   41],
       [  27,  736,    4,  505,    0,    1,   21, 4891,   63,   17],
       [  12,  472,    1,   25,    4,    7,  158,   53, 4218,  901],
       [   3,   74,    0,    6,    9,   43,  418,   17,  315, 5064]], dtype=int32)>
```

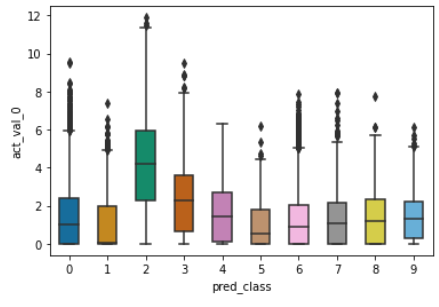
```
# Extracts the outputs of the 2 layers:
layer_outputs = [layer.output for layer in model.layers]

# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

print(f"There are {len(layer_outputs)} layers")
layer_outputs # description of the layers

# Get the outputs of all the hidden nodes for each of the 60000 training images
activations = activation_model.predict(x_train_norm)
hidden_layer_activation = activations[0]
output_layer_activations = activations[1]
hidden_layer_activation.shape   #  each of the 128 hidden nodes has one activation value per training image

output_layer_activations.shape

print(f"The maximum activation value of the hidden nodes in the hidden layer is \
{hidden_layer_activation.max()}")

# Some stats about the output layer as an aside...
np.set_printoptions(suppress = True)  # display probabilities as decimals and NOT in scientific notation
ouput_layer_activation = activations[1]
print(f"The output node has shape {ouput_layer_activation.shape}")
print(f"The output for the first image are {ouput_layer_activation[0].round(4)}")
print(f"The sum of the probabilities is (approximately) {ouput_layer_activation[0].sum()}")
```

```
There are 2 layers
The maximum activation value of the hidden nodes in the hidden layer is 17.98011589050293
The output node has shape (60000, 10)
The output for the first image are [0.    0.    0.    0.002 0.    0.998 0.    0.    0.    0.   ]
The sum of the probabilities is (approximately) 1.0
```

```
#Get the dataframe of all the node values
activation_data = {'pred_class':pred_classes}
for k in range(0,1):
    activation_data[f"act_val_{k}"] = hidden_layer_activation[:,k]
activation_df = pd.DataFrame(activation_data)
activation_df.head()
```

	pred_class	act_val_0
0	6	0.711956
1	0	1.117060
2	6	1.001708
3	1	3.044749
4	9	2.145078

```
# To see how closely the hidden node activation values correlate with the class predictions
# Let us use seaborn for the boxplots this time.
bplot = sns.boxplot(y='act_val_0', x='pred_class',
                 data=activation_df[['act_val_0','pred_class']],
                 width=0.5,
                 palette="colorblind")
```

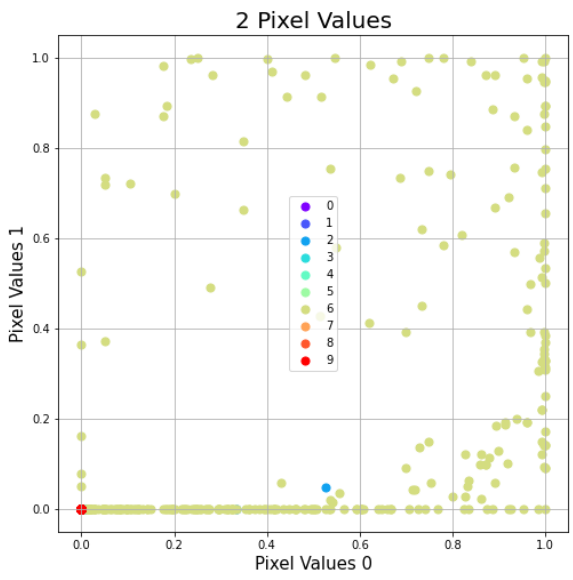



```
pixel_data = {'pred_class':pred_classes}
for k in range(0,784):
    pixel_data[f"pix_val_{k}"] = x_train_norm[:,k]
pixel_df = pd.DataFrame(pixel_data)
pixel_df.head()
```

	pred_class	pix_val_0	pix_val_1	pix_val_2	pix_val_3	pix_val_4	pix_val_5	pix_val_6	pix_val_7	pix_val_8	pix_val_9	pix_val_10	pix_val_11	pix_val_12	pix_val_13	pix_val_14	pix_val_15	pix_val_16	pix_val_17	pix_val_1
0	6	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
1	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
2	6	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
3	1	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
4	9	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

5 rows × 785 columns

```
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Pixel Values 0', fontsize = 15)
ax.set_ylabel('Pixel Values 1', fontsize = 15)
ax.set_title('2 Pixel Values', fontsize = 20)
targets = [0,1,2,3,4,5,6,7,8,9]
#colors = ['r', 'g', 'b']
from matplotlib import cm


colors = cm.rainbow(np.linspace(0, 1, 10))

for target, color in zip(targets,colors):
    indicesToKeep = pixel_df['pred_class'] == target
    ax.scatter(pixel_df.loc[indicesToKeep, 'pix_val_77']
               , pixel_df.loc[indicesToKeep, 'pix_val_78']
               , c = color.reshape(1,-1)
               , s = 50)
ax.legend(targets)
ax.grid()
```



Experiment 2

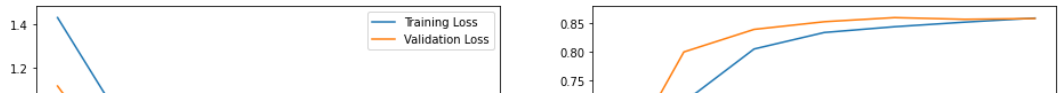
```
model2 = Sequential([
    Dense(input_shape=[784], units = 128, activation = tf.nn.relu),
    Dense(name = "hidden_layer", units = 2, activation = tf.nn.relu),
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])

model2.compile(optimizer='rmsprop',
               loss = 'categorical_crossentropy',
               metrics=['accuracy'])

history = model2.fit(
    x_train_norm
    ,y_train_encoded
    ,epochs = 200
    ,validation_split=0.0833
    ,callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

```
Epoch 1/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.6527 - accuracy: 0.3844 - val_loss: 1.1157 - val_accuracy: 0.5810
Epoch 2/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.0245 - accuracy: 0.6599 - val_loss: 0.6760 - val_accuracy: 0.7999
Epoch 3/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.6627 - accuracy: 0.7933 - val_loss: 0.5148 - val_accuracy: 0.8393
Epoch 4/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.5280 - accuracy: 0.8330 - val_loss: 0.4421 - val_accuracy: 0.8527
Epoch 5/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.4541 - accuracy: 0.8468 - val_loss: 0.4211 - val_accuracy: 0.8599
Epoch 6/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.4131 - accuracy: 0.8518 - val_loss: 0.4044 - val_accuracy: 0.8569
Epoch 7/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3900 - accuracy: 0.8618 - val_loss: 0.3902 - val_accuracy: 0.8583
```

```
loss, accuracy = model2.evaluate(x_test_norm, y_test_encoded)
print('test set accuracy: ', accuracy * 100)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.4583 - accuracy: 0.8444
test set accuracy:  84.43999886512756
```

```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
val_accs = history.history['val_accuracy']
epochs = len(losses)

plt.figure(figsize=(16, 4))
for i, metrics in enumerate(zip([losses, accs], [val_losses, val_accs], ['Loss', 'Accuracy'])):
    plt.subplot(1, 2, i + 1)
    plt.plot(range(epochs), metrics[0], label='Training {}'.format(metrics[2]))
    plt.plot(range(epochs), metrics[1], label='Validation {}'.format(metrics[2]))
    plt.legend()
plt.show()
```




```
pred_classes = np.argmax(model2.predict(x_train_norm), axis=-1)
pred_classes
```

```
array([5, 6, 4, ..., 5, 6, 8])
```



```
conf_mx = tf.math.confusion_matrix(y_train, pred_classes)
conf_mx
```

```
<tf.Tensor: shape=(10, 10), dtype=int32, numpy=
array([[  47,    0,   10,    1,    0,   35, 5814,    0,   16,    0],
       [   0, 6520,  100,    1,   80,    1,    0,    0,   28,   12],
       [   0,   21, 5733,    4,   17,   10,   35,   14,  119,    5],
       [   0,    0,   12, 5717,    0,  210,   10,   77,   99,    6],
       [   0,   24,   59,    0, 5569,    0,    8,    1,   81,  100],
       [  32,    1,    4,  148,    1, 5148,   37,   19,   27,    4],
       [  89,    0,   65,    0,    0,   38, 5715,    0,   11,    0],
       [   0,    2,    9,   97,   12,    5,    3, 5929,   69,  139],
       [   5,    9,   27,   13,   23,  115,   14,   18, 5604,   23],
       [   1,    0,    1,   12,   54,    6,    7,  103,  193, 5572]], dtype=int32)>
```

```
# Extracts the outputs of the 2 layers:
layer_outputs = [layer.output for layer in model.layers]

# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

print(f"There are {len(layer_outputs)} layers")
layer_outputs # description of the layers

# Get the outputs of all the hidden nodes for each of the 60000 training images
activations = activation_model.predict(x_train_norm)
hidden_layer_activation = activations[0]
output_layer_activations = activations[1]
hidden_layer_activation.shape   #  each of the 128 hidden nodes has one activation value per training image

output_layer_activations.shape
```

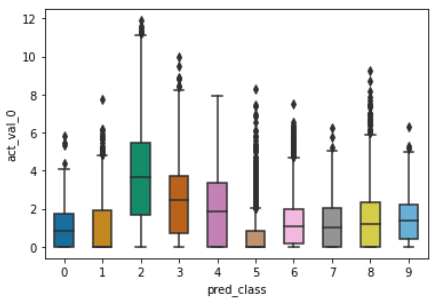
```
print(f"The maximum activation value of the hidden nodes in the hidden layer is \
{hidden_layer_activation.max()}")

# Some stats about the output layer as an aside...
np.set_printoptions(suppress = True)  # display probabilities as decimals and NOT in scientific notation
ouput_layer_activation = activations[1]
print(f"The output node has shape {ouput_layer_activation.shape}")
print(f"The output for the first image are {ouput_layer_activation[0].round(4)}")
print(f"The sum of the probabilities is (approximately) {ouput_layer_activation[0].sum()}")
```

```
There are 2 layers
The maximum activation value of the hidden nodes in the hidden layer is 17.98011589050293
The output node has shape (60000, 10)
The output for the first image are [0.    0.    0.    0.002 0.    0.998 0.    0.    0.    0.   ]
The sum of the probabilities is (approximately) 1.0
```

```
#Get the dataframe of all the node values
activation_data = {'pred_class':pred_classes}
for k in range(0,2):
    activation_data[f"act_val_{k}"] = hidden_layer_activation[:,k]

activation_df = pd.DataFrame(activation_data)
activation_df.head()
```

	pred_class	act_val_0	act_val_1
0	5	0.711956	1.275676
1	6	1.117060	1.126043
2	4	1.001708	1.550730
3	1	3.044749	0.790666
4	9	2.145078	1.221523

```
# To see how closely the hidden node activation values correlate with the class predictions
# Let us use seaborn for the boxplots this time.
bplot = sns.boxplot(y='act_val_0', x='pred_class',
                 data=activation_df[['act_val_0','pred_class']],
                 width=0.5,
                 palette="colorblind")
```



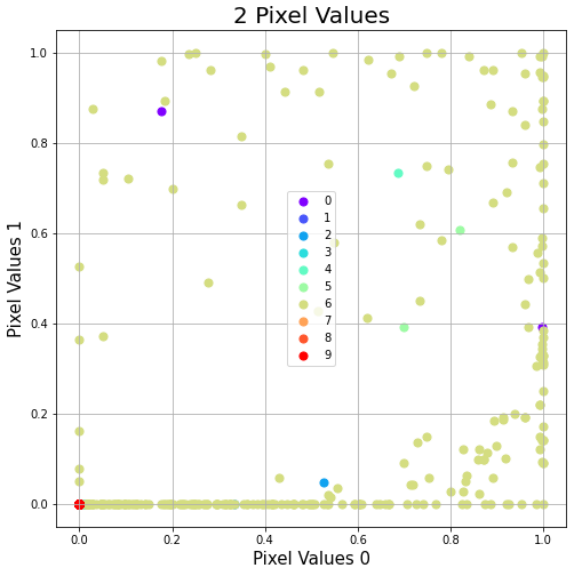

```
pixel_data = {'pred_class':pred_classes}
for k in range(0,784):
    pixel_data[f"pix_val_{k}"] = x_train_norm[:,k]
pixel_df = pd.DataFrame(pixel_data)
pixel_df.head()
```

	pred_class	pix_val_0	pix_val_1	pix_val_2	pix_val_3	pix_val_4	pix_val_5	pix_val_6	pix_val_7	pix_val_8	pix_val_9	pix_val_10	pix_val_11	pix_val_12	pix_val_13	pix_val_14	pix_val_15	pix_val_16	pix_val_17	pix_val_1
0	5	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
1	6	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
2	4	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
3	1	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
4	9	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

5 rows × 785 columns

```
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Pixel Values 0', fontsize = 15)
ax.set_ylabel('Pixel Values 1', fontsize = 15)
ax.set_title('2 Pixel Values', fontsize = 20)
targets = [0,1,2,3,4,5,6,7,8,9]
#colors = ['r', 'g', 'b']
from matplotlib import cm
```

```
colors = cm.rainbow(np.linspace(0, 1, 10))

for target, color in zip(targets,colors):
    indicesToKeep = pixel_df['pred_class'] == target
    ax.scatter(pixel_df.loc[indicesToKeep, 'pix_val_77']
               , pixel_df.loc[indicesToKeep, 'pix_val_78']
               , c = color.reshape(1,-1)
               , s = 50)
ax.legend(targets)
ax.grid()
```



Experiment 3

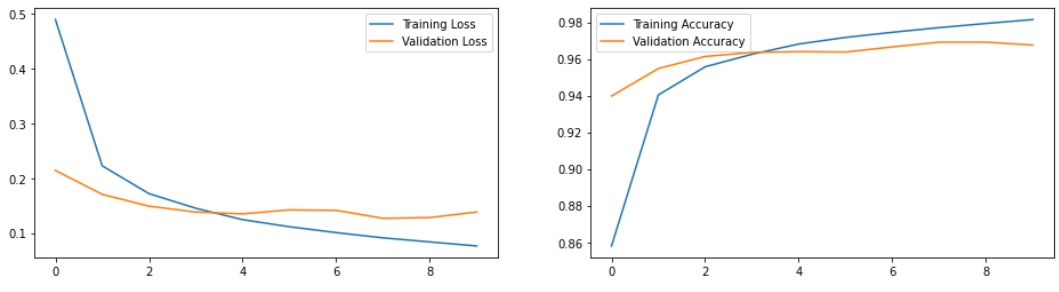
```
model3 = Sequential([
    Dense(input_shape=[784], units = 128, activation = tf.nn.relu),
    Dense(name = "hidden_layer", units = 5, activation = tf.nn.relu),
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])

model3.compile(optimizer='rmsprop',
               loss = 'categorical_crossentropy',
               metrics=['accuracy'])

history = model3.fit(
    x_train_norm
    ,y_train_encoded
    ,epochs = 200
    ,validation_split=0.0833
    ,callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

```
Epoch 1/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.8028 - accuracy: 0.7442 - val_loss: 0.2141 - val_accuracy: 0.9398
Epoch 2/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.2363 - accuracy: 0.9365 - val_loss: 0.1704 - val_accuracy: 0.9548
Epoch 3/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1740 - accuracy: 0.9538 - val_loss: 0.1488 - val_accuracy: 0.9614
Epoch 4/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1475 - accuracy: 0.9621 - val_loss: 0.1381 - val_accuracy: 0.9636
Epoch 5/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1221 - accuracy: 0.9692 - val_loss: 0.1348 - val_accuracy: 0.9640
Epoch 6/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1032 - accuracy: 0.9731 - val_loss: 0.1421 - val_accuracy: 0.9638
Epoch 7/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0963 - accuracy: 0.9756 - val_loss: 0.1411 - val_accuracy: 0.9666
Epoch 8/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0919 - accuracy: 0.9775 - val_loss: 0.1267 - val_accuracy: 0.9692
Epoch 9/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0794 - accuracy: 0.9807 - val_loss: 0.1280 - val_accuracy: 0.9692
Epoch 10/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0696 - accuracy: 0.9829 - val_loss: 0.1382 - val_accuracy: 0.9676
```

```
loss, accuracy = model3.evaluate(x_test_norm, y_test_encoded)
print('test set accuracy: ', accuracy * 100)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.1630 - accuracy: 0.9651
test set accuracy:  96.50999903678894
```

```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
val_accs = history.history['val_accuracy']
epochs = len(losses)

plt.figure(figsize=(16, 4))
for i, metrics in enumerate(zip([losses, accs], [val_losses, val_accs], ['Loss', 'Accuracy'])):
    plt.subplot(1, 2, i + 1)
    plt.plot(range(epochs), metrics[0], label='Training {}'.format(metrics[2]))
    plt.plot(range(epochs), metrics[1], label='Validation {}'.format(metrics[2]))
    plt.legend()
plt.show()
```




```
pred_classes = np.argmax(model3.predict(x_train_norm), axis=-1)
pred_classes
```

```
array([5, 0, 4, ..., 5, 6, 8])
```

```
conf_mx = tf.math.confusion_matrix(y_train, pred_classes)
conf_mx
```

```
<tf.Tensor: shape=(10, 10), dtype=int32, numpy=
array([[5871,    0,   13,    0,    2,    6,    2,    8,   19,    2],
       [   1, 6685,   13,    5,    2,    2,    5,    6,   21,    2],
       [  22,    2, 5885,   17,    4,    4,    0,   11,   12,    1],
```

```
[   0,    3,   15, 6045,    0,   32,    0,   17,   14,    5],
[   3,    8,   17,    0, 5769,    2,    1,    1,    7,   34],
[   4,    0,   11,   55,    4, 5315,    6,    9,    7,   10],
[   5,   10,    0,    9,   13,    9, 5807,    2,   41,   22],
[  22,    4,    8,   20,    1,   15,    2, 6155,   18,   20],
[  10,   11,   29,   52,    2,   10,    0,    7, 5726,    4],
[   8,    1,    1,   11,   33,   68,    2,   42,    9, 5774]], dtype=int32)>
```

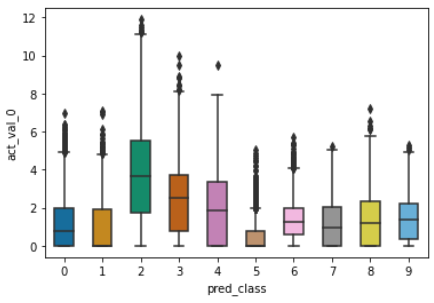
```
# Extracts the outputs of the 2 layers:
layer_outputs = [layer.output for layer in model.layers]

# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

print(f"There are {len(layer_outputs)} layers")
layer_outputs # description of the layers

# Get the outputs of all the hidden nodes for each of the 60000 training images
activations = activation_model.predict(x_train_norm)
hidden_layer_activation = activations[0]
output_layer_activations = activations[1]
hidden_layer_activation.shape   #  each of the 128 hidden nodes has one activation value per training image

output_layer_activations.shape

print(f"The maximum activation value of the hidden nodes in the hidden layer is \
{hidden_layer_activation.max()}")

# Some stats about the output layer as an aside...
np.set_printoptions(suppress = True)  # display probabilities as decimals and NOT in scientific notation
ouput_layer_activation = activations[1]
print(f"The output node has shape {ouput_layer_activation.shape}")
print(f"The output for the first image are {ouput_layer_activation[0].round(4)}")
print(f"The sum of the probabilities is (approximately) {ouput_layer_activation[0].sum()}")
```

```
There are 2 layers
The maximum activation value of the hidden nodes in the hidden layer is 17.98011589050293
The output node has shape (60000, 10)
The output for the first image are [0.    0.    0.    0.002 0.    0.998 0.    0.    0.    0.   ]
The sum of the probabilities is (approximately) 1.0
```

```
#Get the dataframe of all the node values
activation_data = {'pred_class':pred_classes}
for k in range(0,5):
    activation_data[f"act_val_{k}"] = hidden_layer_activation[:,k]

activation_df = pd.DataFrame(activation_data)
activation_df.head()
```

	pred_class	act_val_0	act_val_1	act_val_2	act_val_3	act_val_4
0	5	0.711956	1.275676	0.507729	0.361824	2.179609
1	0	1.117060	1.126043	0.000000	0.000000	0.000000
2	4	1.001708	1.550730	2.589451	1.804716	0.421158
3	1	3.044749	0.790666	1.341971	0.000000	0.000000
4	9	2.145078	1.221523	3.052742	0.000000	0.000000

```
# To see how closely the hidden node activation values correlate with the class predictions
# Let us use seaborn for the boxplots this time.
bplot = sns.boxplot(y='act_val_0', x='pred_class',
                 data=activation_df[['act_val_0','pred_class']],
                 width=0.5,
                 palette="colorblind")
```

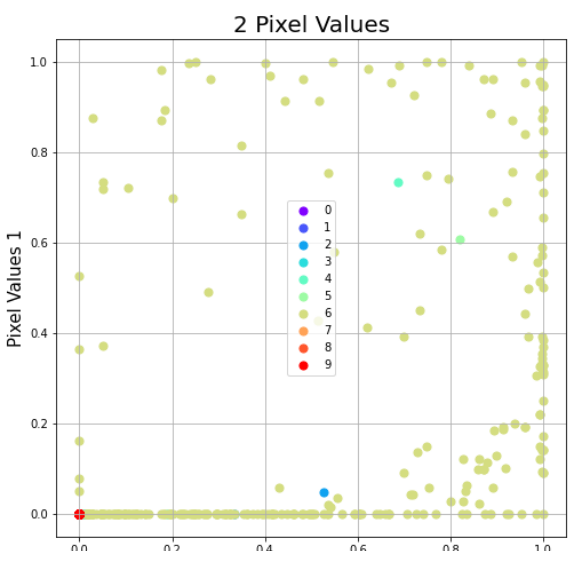



```
pixel_data = {'pred_class':pred_classes}
for k in range(0,784):
    pixel_data[f"pix_val_{k}"] = x_train_norm[:,k]
pixel_df = pd.DataFrame(pixel_data)
pixel_df.head()
```

	pred_class	pix_val_0	pix_val_1	pix_val_2	pix_val_3	pix_val_4	pix_val_5	pix_val_6	pix_val_7	pix_val_8	pix_val_9	pix_val_10	pix_val_11	pix_val_12	pix_val_13	pix_val_14	pix_val_15	pix_val_16	pix_val_17	pix_val_1
0	5	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
1	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
2	4	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
3	1	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
4	9	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

5 rows × 785 columns

```
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Pixel Values 0', fontsize = 15)
ax.set_ylabel('Pixel Values 1', fontsize = 15)
ax.set_title('2 Pixel Values', fontsize = 20)
targets = [0,1,2,3,4,5,6,7,8,9]
#colors = ['r', 'g', 'b']
from matplotlib import cm


colors = cm.rainbow(np.linspace(0, 1, 10))

for target, color in zip(targets,colors):
    indicesToKeep = pixel_df['pred_class'] == target
    ax.scatter(pixel_df.loc[indicesToKeep, 'pix_val_77']
               , pixel_df.loc[indicesToKeep, 'pix_val_78']
               , c = color.reshape(1,-1)
               , s = 50)
ax.legend(targets)
ax.grid()
```



Creating confusion matrices

Let us see what the confusion matrix looks like. Using both `sklearn.metrics` . Then we visualize the confusion matrix and see what that tells us.

```
# Get the predicted classes:
# pred_classes = model.predict_classes(x_train_norm)# give deprecation warning
pred_classes = np.argmax(model.predict(x_train_norm), axis=-1)
pred_classes
```

```
array([5, 0, 4, ..., 5, 6, 8])
```

Correlation matrix that measures the linear relationships
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html

```
conf_mx = tf.math.confusion_matrix(y_train, pred_classes)
conf_mx
```

```
<tf.Tensor: shape=(10, 10), dtype=int32, numpy=
array([[5893,    1,    3,    1,    2,    2,    7,    3,    7,    4],
       [   1, 6722,    3,    1,    3,    0,    1,    3,    8,    0],
       [   6,   60, 5808,   10,    8,    2,    3,    9,   48,    4],
       [   1,    8,   12, 6024,    1,   23,    0,    5,   52,    5],
       [   0,   14,    0,    0, 5813,    1,    5,    2,    0,    7],
       [   1,    4,    2,   13,    3, 5357,   11,    3,   16,   11],
       [   4,    9,    0,    0,    5,    9, 5888,    0,    3,    0],
       [   1,   22,    8,    9,    8,    3,    0, 6181,   11,   22],
       [   4,   10,    2,    7,    2,    7,    5,    2, 5809,    3],
       [   7,    6,    0,   10,   59,   11,    1,   16,   27, 5812]], dtype=int32)>
```

```
print("The first prediction\n {}\n".format(pred_classes[0]))
```

```
The first prediction
 5
```

```
print("First ten entries of the predictions:\n {}\n".format(pred_classes[0:10]))
```

```
First ten entries of the predictions:
 [5 0 4 1 9 2 1 3 1 4]
```

```
cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)
```

```
df = pd.DataFrame(preds[0:20], columns = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'])
df.style.format("{:.2%}").background_gradient(cmap=cm)
```

	0	1	2	3	4	5	6	7	8	9
0	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%	0.00%	0.00%
1	0.00%	0.00%	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
2	0.00%	99.98%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.02%	0.00%
3	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
4	0.00%	0.00%	0.00%	0.00%	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%
5	0.00%	99.99%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
6	0.00%	0.00%	0.00%	0.00%	99.98%	0.00%	0.00%	0.00%	0.01%	0.00%
7	0.00%	0.00%	0.00%	1.97%	0.01%	0.00%	0.00%	0.00%	0.00%	98.02%
8	0.00%	0.00%	0.00%	0.00%	0.00%	99.10%	0.90%	0.00%	0.00%	0.00%
9	0.00%	0.00%	0.00%	0.00%	0.04%	0.00%	0.00%	0.00%	0.00%	99.96%
10	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
11	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%	0.00%	0.00%	0.00%
12	0.00%	0.00%	0.00%	0.00%	0.05%	0.00%	0.00%	0.00%	0.00%	99.95%
13	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
14	0.00%	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
15	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%	0.00%	0.00%	0.00%	0.00%
16	0.00%	0.00%	0.00%	0.00%	0.01%	0.00%	0.00%	0.00%	0.09%	99.90%
17	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%	0.00%	0.00%

Visualizing the confusion matrix

We use code from chapter 3 of Hands on Machine Learning (A. Geron) (cf. https://github.com/ageron/handson-ml2/blob/master/03_classification.ipynb) to display a "heat map" of the confusion matrix. Then we normalize the confusion matrix so we can compare error rates.

See https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch03.html#classification_chapter

```
[ ]
```
16 cells hidden

Analyzing the activation values of the hidden nodes

We want to examine the contribution of the individual hidden nodes to the classifications made by the model. We first get the activation values of all the hidden nodes for each of the 60,000 training images and treat these *128 activations* as the features that determine the classification class. For the sake of comparison, we also consider the *784 pixels* of each training image and determine the contribution of the individual pixels to the predicted classification class.

Our goal is to use *box* and *scatter* plots to visualize how these features (*pixel* and *activiation* values) correlate with the predicted classes. Because of the high dimension of the feature spaces, we apply *PCA decomposition* and *t-Distributed stochastic neighbor embedding* (`t-SNE`) to reduce the number of features in each case.

We use the following two articles as reference

- https://towardsdatascience.com/pca-using-python-scikit-learn-e653f8989e60
- https://towardsdatascience.com/visualising-high-dimensional-datasets-using-pca-and-t-sne-in-python-8ef87e7915b

1) Raw data is 60,000 X 784. Just do a scatter plot of col 1 vs col 2. Overlay the color coded classes. We should not see any patterns since there is not much info in 2 cols to discriminate.

2) PCA of raw data – as we discussed earlier. Plot PC1 vs PC2 with overlay. This should be 'better' since these 2 capture the info from all 784 cols.

3) PCA of activation values – as we discussed earlier. This should be 'better' than the previous 2 since it has captured specific features of discrimination.

Getting the activation values of the hidden nodes

To get the activation values of the hidden nodes, we need to create a new model, `activation_model`, that takes the same input as our current model but outputs the activation value of the hidden layer, i.e. of the hidden node. Then use the `predict` function to get the activation values.

```
# Extracts the outputs of the 2 layers:
layer_outputs = [layer.output for layer in model.layers]

# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

print(f"There are {len(layer_outputs)} layers")
layer_outputs # description of the layers
```

```
There are 2 layers
[<KerasTensor: shape=(None, 128) dtype=float32 (created by layer 'dense')>,
 <KerasTensor: shape=(None, 10) dtype=float32 (created by layer 'output_layer')>]
```

```
# Get the outputs of all the hidden nodes for each of the 60000 training images
activations = activation_model.predict(x_train_norm)
hidden_layer_activation = activations[0]
output_layer_activations = activations[1]
hidden_layer_activation.shape   #  each of the 128 hidden nodes has one activation value per training image
```

```
(60000, 128)
```

```
output_layer_activations.shape
```

```
(60000, 10)
```

```
print(f"The maximum activation value of the hidden nodes in the hidden layer is \
{hidden_layer_activation.max()}")
```

```
The maximum activation value of the hidden nodes in the hidden layer is 17.98011589050293
```

```
# Some stats about the output layer as an aside...
np.set_printoptions(suppress = True)  # display probabilities as decimals and NOT in scientific notation
ouput_layer_activation = activations[1]
print(f"The output node has shape {ouput_layer_activation.shape}")
print(f"The output for the first image are {ouput_layer_activation[0].round(4)}")
print(f"The sum of the probabilities is (approximately) {ouput_layer_activation[0].sum()}")
```

```
The output node has shape (60000, 10)
The output for the first image are [0.    0.    0.    0.002 0.    0.998 0.    0.    0.    0.   ]
The sum of the probabilities is (approximately) 1.0
```

Creating a dataframe with the activation values and predicted classes

```
#Get the dataframe of all the node values
activation_data = {'pred_class':pred_classes}
for k in range(0,128):
    activation_data[f"act_val_{k}"] = hidden_layer_activation[:,k]

activation_df = pd.DataFrame(activation_data)
activation_df.head()
```

	pred_class	act_val_0	act_val_1	act_val_2	act_val_3	act_val_4	act_val_5	act_val_6	act_val_7	act_val_8	act_val_9	act_val_10	act_val_11	act_val_12	act_val_13	act_val_14	act_val_15	act_val_16	act_val_17	act_val_1
0	5	0.711956	1.275676	0.507729	0.361824	2.179609	0.000000	4.236779	1.191710	0.000000	1.556238	0.000000	1.450128	0.970069	0.000000	0.502491	3.926322	0.929150	0.842654	0.00000
1	0	1.117060	1.126043	0.000000	0.000000	0.000000	2.989175	0.000000	0.594822	0.000000	0.000000	0.000000	0.000000	1.223132	1.065335	0.000000	1.466588	0.000000	0.000000	0.00000
2	4	1.001708	1.550730	2.589451	1.804716	0.421158	2.719532	0.000000	1.556091	0.000000	0.000000	0.719144	0.000000	0.000000	1.856879	0.000000	0.000000	0.000000	2.346549	0.00000
3	1	3.044749	0.790666	1.341971	0.000000	0.000000	1.052578	5.280775	0.000000	0.390522	0.000000	1.460953	0.055368	3.122771	0.135974	0.000000	0.000000	0.000000	2.579069	0.00000
4	9	2.145078	1.221523	3.052742	0.000000	0.000000	3.714122	0.063753	0.000000	1.373083	0.000000	5.516031	0.000000	0.000000	1.893915	0.689452	0.082704	2.334632	0.892361	1.76157

5 rows × 129 columns

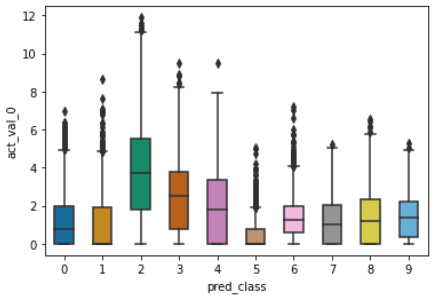
Visualualizing the activation values with boxplots

We get the activation values of the first hidden node and combine them with the corresponding predicted classes into a DataFrame. We use both `matplotlib` and `seaborn` to create boxplots from the dataframe.

Correlation matrix that measures the linear relationships
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html

Correlation matrix that measures the linear relationships
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html

```
# To see how closely the hidden node activation values correlate with the class predictions
# Let us use seaborn for the boxplots this time.
bplot = sns.boxplot(y='act_val_0', x='pred_class',
                    data=activation_df[['act_val_0','pred_class']],
                    width=0.5,
                    palette="colorblind")
```



Creating a dataframe with the pixel values and predicted classes

```
#Get the dataframe of all the pixel values
pixel_data = {'pred_class':pred_classes}
for k in range(0,784):
    pixel_data[f"pix_val_{k}"] = x_train_norm[:,k]
pixel_df = pd.DataFrame(pixel_data)
pixel_df.head()
```

	pred_class	pix_val_0	pix_val_1	pix_val_2	pix_val_3	pix_val_4	pix_val_5	pix_val_6	pix_val_7	pix_val_8	pix_val_9	pix_val_10	pix_val_11	pix_val_12	pix_val_13	pix_val_14	pix_val_15	pix_val_16	pix_val_17	pix_val_1
0	5	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
1	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
2	4	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
3	1	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
4	9	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

5 rows × 785 columns

```
pixel_df.pix_val_77.value_counts()
```

```
0.000000    59720
1.000000       25
0.996078       13
0.992157        9
0.050980        6
            ...
0.392157        1
0.717647        1
0.215686        1
0.925490        1
0.937255        1
Name: pix_val_77, Length: 150, dtype: int64
```

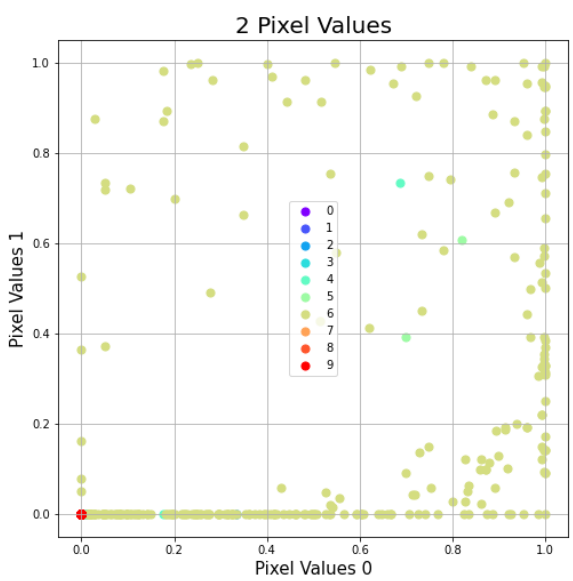
```
pixel_df.pix_val_78.value_counts()
```

```
0.000000    59862
1.000000        6
0.141176        4
0.960784        4
0.992157        4
            ...
0.749020        1
0.717647        1
0.345098        1
0.968627        1
0.654902        1
Name: pix_val_78, Length: 97, dtype: int64
```

Using a scatter plot to visualize the contribution of two pixel location values to the predicated classes

We use a scatter plot to determine the correlation between the `pix_val_77` and `pix_val_78` values and the `pred_class` values.

```
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Pixel Values 0', fontsize = 15)
ax.set_ylabel('Pixel Values 1', fontsize = 15)
ax.set_title('2 Pixel Values', fontsize = 20)
targets = [0,1,2,3,4,5,6,7,8,9]
#colors = ['r', 'g', 'b']
from matplotlib import cm
colors = cm.rainbow(np.linspace(0, 1, 10))

for target, color in zip(targets,colors):
    indicesToKeep = pixel_df['pred_class'] == target
    ax.scatter(pixel_df.loc[indicesToKeep, 'pix_val_77']
               , pixel_df.loc[indicesToKeep, 'pix_val_78']
               , c = color.reshape(1,-1)
               , s = 50)
ax.legend(targets)
ax.grid()
```



PCA Feature Reduction / Model Optimization

Correlation matrix that measures the linear relationships

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html

Use PCA decomposition to reduce the number of features from 784 features to 2 features

```
# from sklearn.decomposition import PCA

# Separating out the features
features = [*pixel_data][1:] # ['pix_val_0', 'pix_val_1',...]
x = pixel_df.loc[:, features].values

pca = PCA(n_components=2)
principalComponents = pca.fit_transform(x)
principalDf = pd.DataFrame(data = principalComponents
             , columns = ['principal component 1', 'principal component 2'])

pixel_pca_df = pd.concat([principalDf, pixel_df[['pred_class']]], axis = 1)

pixel_pca_df.head()
```

	principal component 1	principal component 2	pred_class
0	0.486040	-1.226358	5
1	3.967556	-1.156220	0
2	-0.203327	1.537845	4
3	-3.133843	-2.381269	1
4	-1.501006	2.865016	9

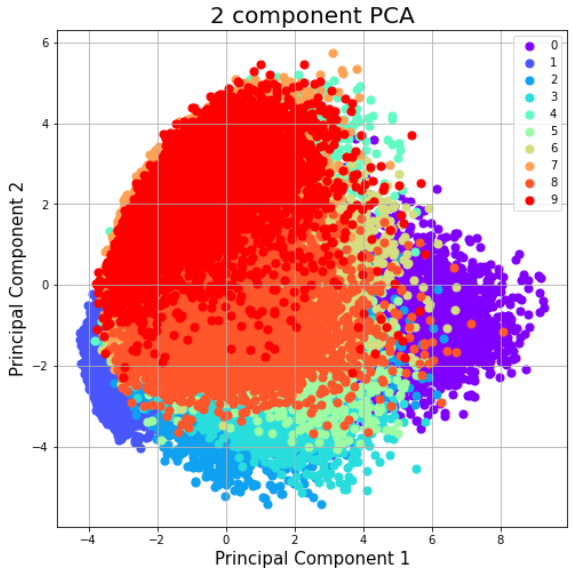
```
pca.explained_variance_ratio_
```

```
array([0.097, 0.071], dtype=float32)
```

Using a scatter plot to visualize the contribution of two principal component values to the predicated class values

```
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Principal Component 1', fontsize = 15)
ax.set_ylabel('Principal Component 2', fontsize = 15)
```

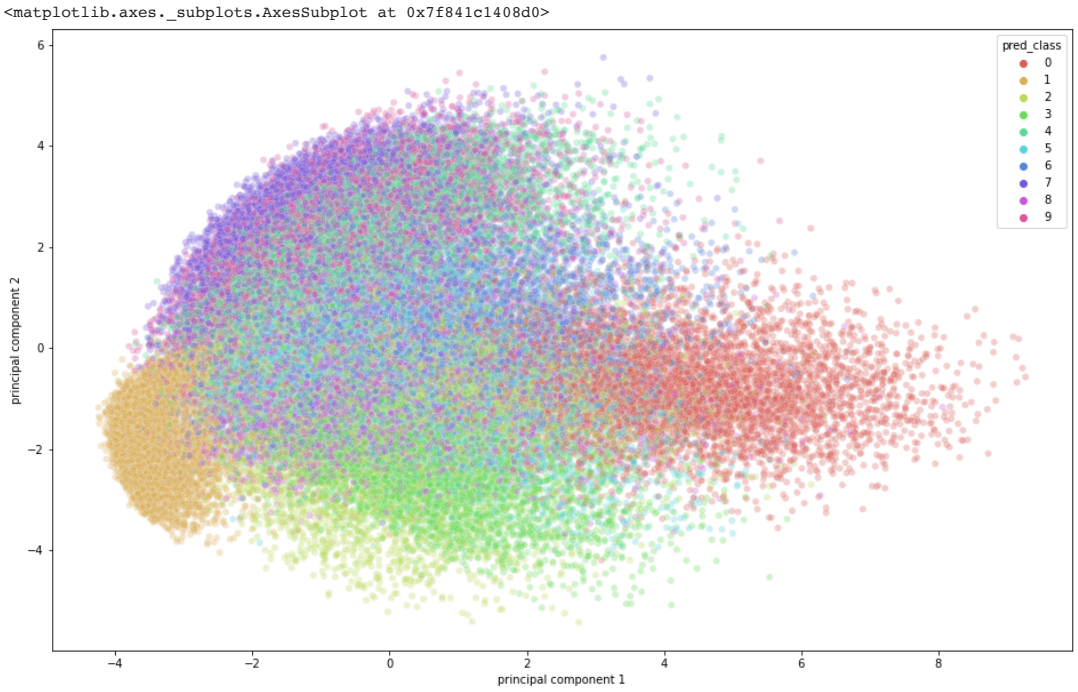
```
ax.set_title('2 component PCA', fontsize = 20)
targets = [0,1,2,3,4,5,6,7,8,9]
#colors = ['r', 'g', 'b']
from matplotlib import cm


colors = cm.rainbow(np.linspace(0, 1, 10))

for target, color in zip(targets,colors):
    indicesToKeep = pixel_pca_df['pred_class'] == target
    ax.scatter(pixel_pca_df.loc[indicesToKeep, 'principal component 1']
               , pixel_pca_df.loc[indicesToKeep, 'principal component 2']
               , c = color.reshape(1,-1)
               , s = 50)
ax.legend(targets)
ax.grid()
```



Using seaborn this time...

```
plt.figure(figsize=(16,10))
sns.scatterplot(
    x="principal component 1", y="principal component 2",
    hue="pred_class",
    palette=sns.color_palette("hls", 10),
    data=pixel_pca_df,
    legend="full",
    alpha=0.3
)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f841c1408d0>
```



Use PCA decomposition to reduce the (activation) features from 128 (= num of hidden nodes) to 2

```
# Separating out the features
features = [*activation_data][1:] # ['act_val_0', 'act_val_1',...]
x = activation_df.loc[:, features].values

pca = PCA(n_components=2)
principalComponents = pca.fit_transform(x)
principalDf = pd.DataFrame(data = principalComponents
             , columns = ['principal component 1', 'principal component 2'])
principalDf.head()
```

	principal component 1	principal component 2
0	0.081705	-6.414558
1	12.448021	2.420921
2	2.813413	5.201393
3	-3.868423	-3.144666
4	-6.306571	6.416787

```
activation_pca_df = pd.concat([principalDf, activation_df[['pred_class']]], axis = 1)
activation_pca_df.head()
```

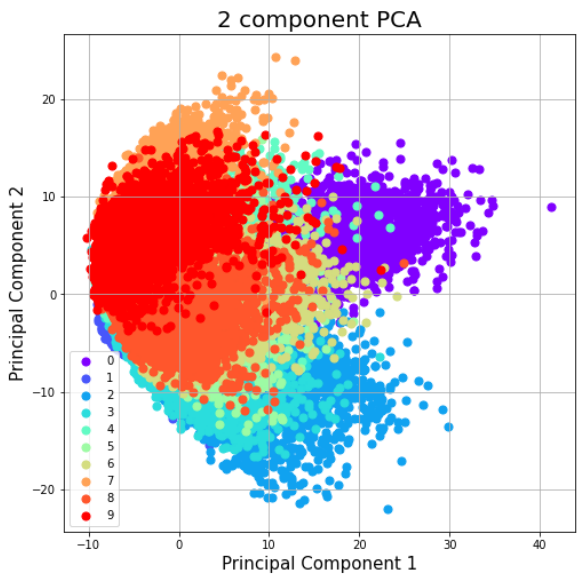
	principal component 1	principal component 2	pred_class
0	0.081705	-6.414558	5
1	12.448021	2.420921	0
2	2.813413	5.201393	4
3	-3.868423	-3.144666	1
4	-6.306571	6.416787	9

```
pca.explained_variance_ratio_
```

```
array([0.169, 0.112], dtype=float32)
```

Using a scatter plot to visualize the contribution of *two* principal component values to the predicated class values

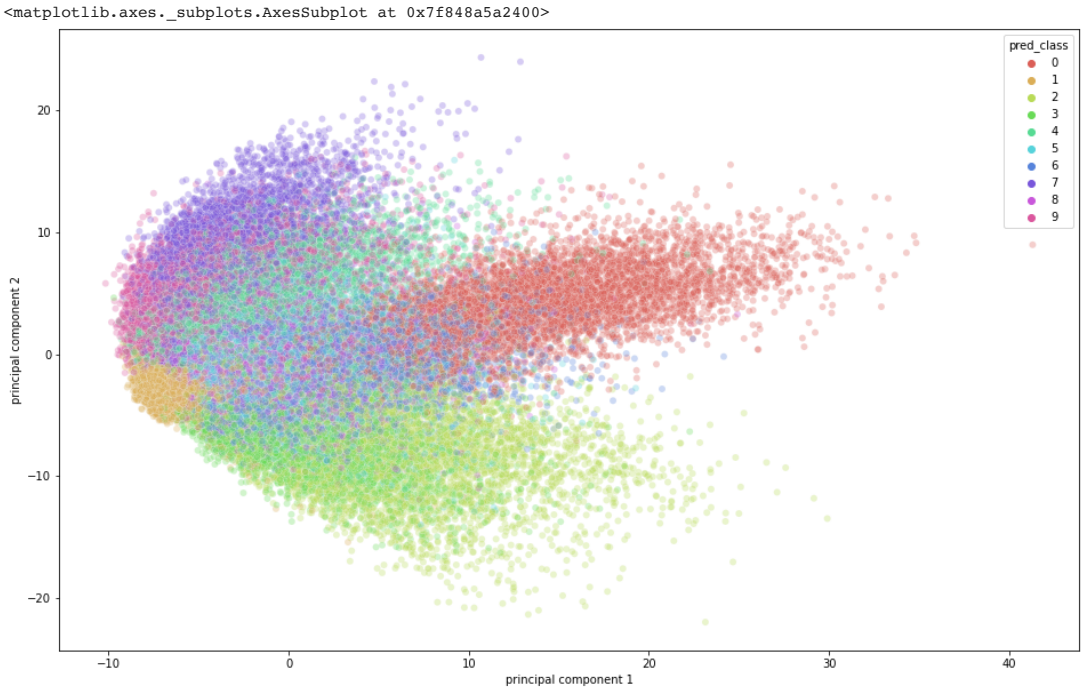
```
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Principal Component 1', fontsize = 15)
ax.set_ylabel('Principal Component 2', fontsize = 15)
ax.set_title('2 component PCA', fontsize = 20)
targets = [0,1,2,3,4,5,6,7,8,9]
#colors = ['r', 'g', 'b']
from matplotlib import cm


colors = cm.rainbow(np.linspace(0, 1, 10))

for target, color in zip(targets,colors):
    indicesToKeep = activation_pca_df['pred_class'] == target
    ax.scatter(activation_pca_df.loc[indicesToKeep, 'principal component 1']
               , activation_pca_df.loc[indicesToKeep, 'principal component 2']
               , c = color.reshape(1,-1)
               , s = 50)
ax.legend(targets)
ax.grid()
```



Using seaborn this time

```
plt.figure(figsize=(16,10))
sns.scatterplot(
    x="principal component 1", y="principal component 2",
    hue="pred_class",
    palette=sns.color_palette("hls", 10),
    data=activation_pca_df,
    legend="full",
    alpha=0.3
)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f848a5a2400>
```



Use PCA decomposition to reduce the (activation) features from 128 (= num of hidden nodes) to 3

```
# Separating out the features
features = [*activation_data][1:] # ['act_val_0', 'act_val_1',...]
x = activation_df.loc[:, features].values

pca = PCA(n_components=3)
principalComponents = pca.fit_transform(x)
principalDf = pd.DataFrame(data = principalComponents
             , columns = ['pca-one', 'pca-two', 'pca-three'])
principalDf.head()
```

	pca-one	pca-two	pca-three
0	0.081682	-6.414784	6.878499
1	12.447898	2.420907	-3.023814
2	2.813410	5.201391	-2.426407
3	-3.868424	-3.144653	-4.674575
4	-6.306576	6.416770	-0.659563

```
pca.explained_variance_ratio_
```

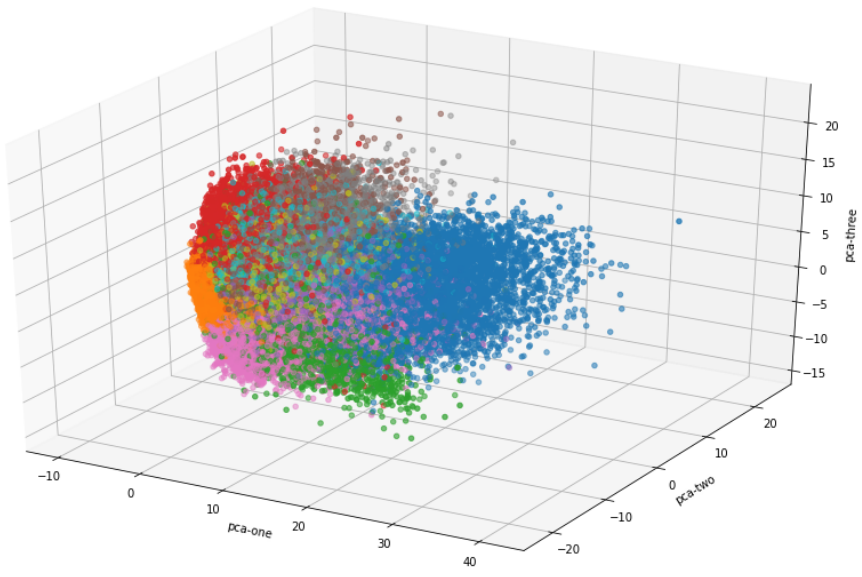
```
array([0.169, 0.112, 0.09 ], dtype=float32)
```

```
activation_pca_df = pd.concat([principalDf, activation_df[['pred_class']]], axis = 1)
activation_pca_df.head()
```

	pca-one	pca-two	pca-three	pred_class
0	0.081682	-6.414784	6.878499	5
1	12.447898	2.420907	-3.023814	0
2	2.813410	5.201391	-2.426407	4
3	-3.868424	-3.144653	-4.674575	1
4	-6.306576	6.416770	-0.659563	9

Using a scatter plot to visualize the contribution of *three* principal component values to the predicated class values

```
# uncomment to to able to rotate the graph...
# %matplotlib notebook
ax = plt.figure(figsize=(16,10)).gca(projection='3d')
ax.scatter(
    xs=activation_pca_df.loc[:,"pca-one"],
    ys=activation_pca_df.loc[:,"pca-two"],
    zs=activation_pca_df.loc[:,"pca-three"],
    c=activation_pca_df.loc[:,"pred_class"],
    cmap='tab10'
)
ax.set_xlabel('pca-one')
ax.set_ylabel('pca-two')
ax.set_zlabel('pca-three')
plt.show()
```



Experiment 4

```
# from sklearn.decomposition import PCA

# Separating out the features
features = [*pixel_data][1:] # ['pix_val_0', 'pix_val_1',...]
x = pixel_df.loc[:, features].values

pca = PCA(n_components= 154)
principalComponents = pca.fit_transform(x)
principalDf = pd.DataFrame(data = principalComponents
             , columns = ['principal component ' + str(i) for i in range(1, 155)])
principalDf.head()
```

	principal component 1	principal component 2	principal component 3	principal component 4	principal component 5	principal component 6	principal component 7	principal component 8	principal component 9	principal component 10	principal component 11	principal component 12	principal component 13	principal component 14	principal component 15	principal component 16	principal component 17	principal component 18	principal component 19	principal component 20	principa... componen... 2
0	0.485966	-1.226145	-0.096110	-2.179480	-0.107081	-0.911641	0.917643	0.626625	-1.425583	0.778151	0.774519	-0.996235	-0.445138	2.938407	0.859823	-0.018336	1.294745	1.212350	1.088480	0.652335	0.10810...
1	3.967518	-1.156323	2.338622	-1.806903	-3.244226	-0.713562	-0.176572	-0.411631	0.158655	0.592009	-1.123520	-0.420909	-1.251741	0.356443	-0.932821	-0.635034	-0.211938	0.155571	0.204578	-0.400137	0.82568...
2	-0.203331	1.537955	-0.739275	2.043197	-1.202643	-0.007189	-3.368838	1.445452	-0.449219	-0.700040	1.766822	-0.623100	0.733262	0.428181	-0.567885	-0.750150	0.665076	-0.479447	-0.036243	-1.703624	-0.24151...
3	-3.133825	-2.381161	1.073143	0.415207	-0.007260	2.743734	-1.857704	-0.263991	1.187180	0.043588	-1.695951	-0.686410	0.956461	0.649793	-0.594949	0.199010	0.040110	0.755516	0.551582	0.518484	-0.25458...
4	-1.501007	2.864861	0.064139	-0.947854	0.384940	0.169553	-0.359483	-1.590418	0.884299	0.408290	-1.363748	1.802600	-1.004629	-1.232302	0.232631	-1.073192	0.075077	-0.705617	0.256127	-0.980633	-0.02669...

5 rows × 154 columns

```
pixel_pca_df = pd.concat([principalDf, pixel_df[['pred_class']]], axis = 1)
pixel_pca_df.head()
from sklearn.model_selection import train_test_split
X_trainpca, X_testpca, y_trainpca, y_testpca  = train_test_split(principalDf, y_train_encoded, test_size=0.20)


model4 = Sequential([
    Dense(input_shape=[154], units = 128, activation = tf.nn.relu),
    Dense(name = "hidden_layer", units = 5, activation = tf.nn.relu),
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])

model4.compile(optimizer='rmsprop',
               loss = 'categorical_crossentropy',
               metrics=['accuracy'])

history = model4.fit(
     X_trainpca
    ,y_trainpca
    ,epochs = 200
    ,validation_split=0.0833
    ,callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

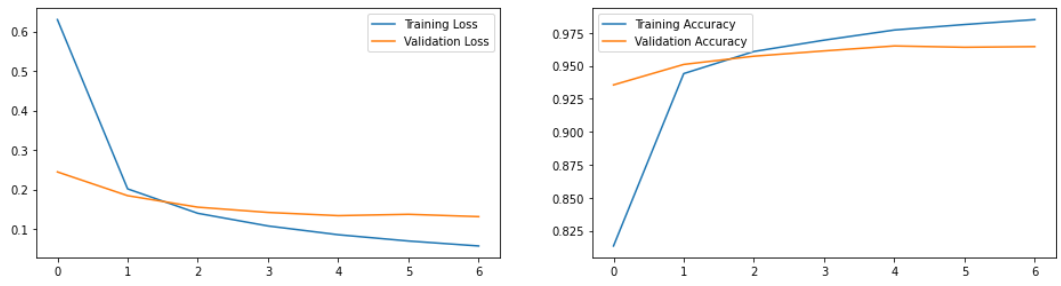
```
Epoch 1/200
1376/1376 [==============================] - 4s 3ms/step - loss: 1.1026 - accuracy: 0.6480 - val_loss: 0.2445 - val_accuracy: 0.9355
Epoch 2/200
1376/1376 [==============================] - 4s 3ms/step - loss: 0.2181 - accuracy: 0.9402 - val_loss: 0.1844 - val_accuracy: 0.9510
Epoch 3/200
1376/1376 [==============================] - 4s 3ms/step - loss: 0.1483 - accuracy: 0.9593 - val_loss: 0.1552 - val_accuracy: 0.9572
Epoch 4/200
1376/1376 [==============================] - 4s 3ms/step - loss: 0.1071 - accuracy: 0.9707 - val_loss: 0.1419 - val_accuracy: 0.9612
Epoch 5/200
1376/1376 [==============================] - 4s 3ms/step - loss: 0.0849 - accuracy: 0.9778 - val_loss: 0.1338 - val_accuracy: 0.9650
Epoch 6/200
1376/1376 [==============================] - 4s 3ms/step - loss: 0.0672 - accuracy: 0.9825 - val_loss: 0.1372 - val_accuracy: 0.9640
```

```
Epoch 7/200
1376/1376 [==============================] - 4s 3ms/step - loss: 0.0540 - accuracy: 0.9863 - val_loss: 0.1313 - val_accuracy: 0.9645
```

```
loss, accuracy = model4.evaluate(X_testpca, y_testpca)
print('test set accuracy: ', accuracy * 100)
```

```
375/375 [==============================] - 1s 2ms/step - loss: 0.1380 - accuracy: 0.9624
test set accuracy:  96.24166488647461
```

```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
val_accs = history.history['val_accuracy']
epochs = len(losses)

plt.figure(figsize=(16, 4))
for i, metrics in enumerate(zip([losses, accs], [val_losses, val_accs], ['Loss', 'Accuracy'])):
    plt.subplot(1, 2, i + 1)
    plt.plot(range(epochs), metrics[0], label='Training {}'.format(metrics[2]))
    plt.plot(range(epochs), metrics[1], label='Validation {}'.format(metrics[2]))
    plt.legend()
plt.show()
```




```
pred_classes = np.argmax(model4.predict(principalComponents), axis=-1)
pred_classes
```

```
array([5, 0, 4, ..., 5, 6, 8])
```

```
conf_mx = tf.math.confusion_matrix(y_train, pred_classes)
conf_mx
```

```
<tf.Tensor: shape=(10, 10), dtype=int32, numpy=
array([[5867,    3,    6,   14,    1,   10,   14,    1,    5,    2],
       [   1, 6659,   29,   15,    4,    0,    1,    9,   23,    1],
       [   9,   13, 5825,   56,   18,    0,    9,   15,    4,    9],
       [   3,    2,   21, 6031,    0,   24,    3,    3,   27,   17],
       [   0,    1,   28,    1, 5753,    1,    5,    3,   11,   39],
       [  10,    1,    1,   47,    2, 5313,   12,    0,   21,   14],
       [   7,    4,    2,    2,    7,   14, 5867,    0,   15,    0],
       [   6,   18,   48,    3,   15,    2,    0, 6133,    0,   40],
       [   4,   14,    2,   54,    9,   18,   10,    2, 5721,   17],
       [   5,    2,    6,   26,   20,   23,    0,   18,   19, 5830]], dtype=int32)>
```

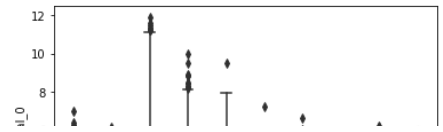
```
# Extracts the outputs of the 2 layers:
layer_outputs = [layer.output for layer in model.layers]

# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

print(f"There are {len(layer_outputs)} layers")
layer_outputs # description of the layers

# Get the outputs of all the hidden nodes for each of the 60000 training images
activations = activation_model.predict(x_train_norm)
hidden_layer_activation = activations[0]
output_layer_activations = activations[1]
hidden_layer_activation.shape   #  each of the 128 hidden nodes has one activation value per training image

output_layer_activations.shape

print(f"The maximum activation value of the hidden nodes in the hidden layer is \
{hidden_layer_activation.max()}")

# Some stats about the output layer as an aside...
np.set_printoptions(suppress = True)  # display probabilities as decimals and NOT in scientific notation
ouput_layer_activation = activations[1]
print(f"The output node has shape {ouput_layer_activation.shape}")
print(f"The output for the first image are {ouput_layer_activation[0].round(4)}")
print(f"The sum of the probabilities is (approximately) {ouput_layer_activation[0].sum()}")
```

```
There are 2 layers
The maximum activation value of the hidden nodes in the hidden layer is 17.98011589050293
The output node has shape (60000, 10)
The output for the first image are [0.    0.    0.    0.002 0.    0.998 0.    0.    0.    0.   ]
The sum of the probabilities is (approximately) 1.0
```

```
#Get the dataframe of all the node values
activation_data = {'pred_class':pred_classes}
for k in range(0,5):
    activation_data[f"act_val_{k}"] = hidden_layer_activation[:,k]

activation_df = pd.DataFrame(activation_data)
activation_df.head()
```

	pred_class	act_val_0	act_val_1	act_val_2	act_val_3	act_val_4
0	5	0.711956	1.275676	0.507729	0.361824	2.179609
1	0	1.117060	1.126043	0.000000	0.000000	0.000000
2	4	1.001708	1.550730	2.589451	1.804716	0.421158
3	1	3.044749	0.790666	1.341971	0.000000	0.000000
4	9	2.145078	1.221523	3.052742	0.000000	0.000000

```
# To see how closely the hidden node activation values correlate with the class predictions
# Let us use seaborn for the boxplots this time.
bplot = sns.boxplot(y='act_val_0', x='pred_class',
                 data=activation_df[['act_val_0','pred_class']],
                 width=0.5,
                 palette="colorblind")
```

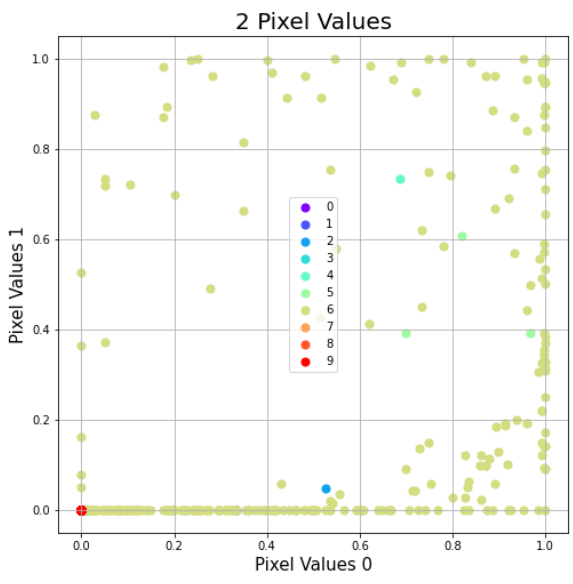


```
pixel_data = {'pred_class':pred_classes}
for k in range(0,784):
    pixel_data[f"pix_val_{k}"] = x_train_norm[:,k]
pixel_df = pd.DataFrame(pixel_data)
pixel_df.head()
```

	pred_class	pix_val_0	pix_val_1	pix_val_2	pix_val_3	pix_val_4	pix_val_5	pix_val_6	pix_val_7	pix_val_8	pix_val_9	pix_val_10	pix_val_11	pix_val_12	pix_val_13	pix_val_14	pix_val_15	pix_val_16	pix_val_17	pix_val_1
0	5	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
1	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
2	4	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
3	1	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
4	9	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

5 rows × 785 columns

```
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Pixel Values 0', fontsize = 15)
ax.set_ylabel('Pixel Values 1', fontsize = 15)
ax.set_title('2 Pixel Values', fontsize = 20)
targets = [0,1,2,3,4,5,6,7,8,9]
#colors = ['r', 'g', 'b']
from matplotlib import cm


colors = cm.rainbow(np.linspace(0, 1, 10))

for target, color in zip(targets,colors):
    indicesToKeep = pixel_df['pred_class'] == target
    ax.scatter(pixel_df.loc[indicesToKeep, 'pix_val_77']
               , pixel_df.loc[indicesToKeep, 'pix_val_78']
               , c = color.reshape(1,-1)
               , s = 50)
ax.legend(targets)
ax.grid()
```



Use t-Distributed Stochastic Neighbor Embedding (**t-SNE**) to reduce the (activation) features from 128 (= num of hidden nodes) to 2

`t-Distributed Stochastic Neighbor Embedding` (**t-SNE**) is another technique for dimensionality reduction and is particularly well suited for the visualization of high-dimensional datasets. This time we only use the first 10,000 training images (N=10000) since the technique is computationally expensive.

See http://jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf

Correlation matrix that measures the linear relationships
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html

```
# Separating out the features
features = [*activation_data][1:] # ['act_val_0', 'act_val_1',...]
x = activation_df.loc[:, features].values

pca = PCA(n_components=2)
principalComponents = pca.fit_transform(x)
principalDf = pd.DataFrame(data = principalComponents
             , columns = ['principal component 1', 'principal component 2'])
principalDf.head()

activation_pca_df = pd.concat([principalDf, activation_df[['pred_class']]], axis = 1)
activation_pca_df.head()

#Get the dataframe of all the node values
activation_data = {'pred_class':pred_classes}
for k in range(0,128):
    activation_data[f"act_val_{k}"] = hidden_layer_activation[:,k]

activation_df = pd.DataFrame(activation_data)
activation_df.head()
```

	pred_class	act_val_0	act_val_1	act_val_2	act_val_3	act_val_4	act_val_5	act_val_6	act_val_7	act_val_8	act_val_9	act_val_10	act_val_11	act_val_12	act_val_13	act_val_14	act_val_15	act_val_16	act_val_17	act_val_1
0	5	0.711956	1.275676	0.507729	0.361824	2.179609	0.000000	4.236779	1.191710	0.000000	1.556238	0.000000	1.450128	0.970069	0.000000	0.502491	3.926322	0.929150	0.842654	0.00000
1	0	1.117060	1.126043	0.000000	0.000000	0.000000	2.989175	0.000000	0.594822	0.000000	0.000000	0.000000	0.000000	1.223132	1.065335	0.000000	1.466588	0.000000	0.000000	0.00000
2	4	1.001708	1.550730	2.589451	1.804716	0.421158	2.719532	0.000000	1.556091	0.000000	0.000000	0.719144	0.000000	0.000000	1.856879	0.000000	0.000000	0.000000	2.346549	0.00000
3	1	3.044749	0.790666	1.341971	0.000000	0.000000	1.052578	5.280775	0.000000	0.390522	0.000000	1.460953	0.055368	3.122771	0.135974	0.000000	0.000000	0.000000	2.579069	0.00000
4	9	2.145078	1.221523	3.052742	0.000000	0.000000	3.714122	0.063753	0.000000	1.373083	0.000000	5.516031	0.000000	0.000000	1.893915	0.689452	0.082704	2.334632	0.892361	1.76157

5 rows × 129 columns

```
N=10000
activation_df_subset = activation_df.iloc[:N].copy()
activation_df_subset.shape
```

```
(10000, 129)
```

```
data_subset = activation_df_subset[features].values
data_subset.shape
```

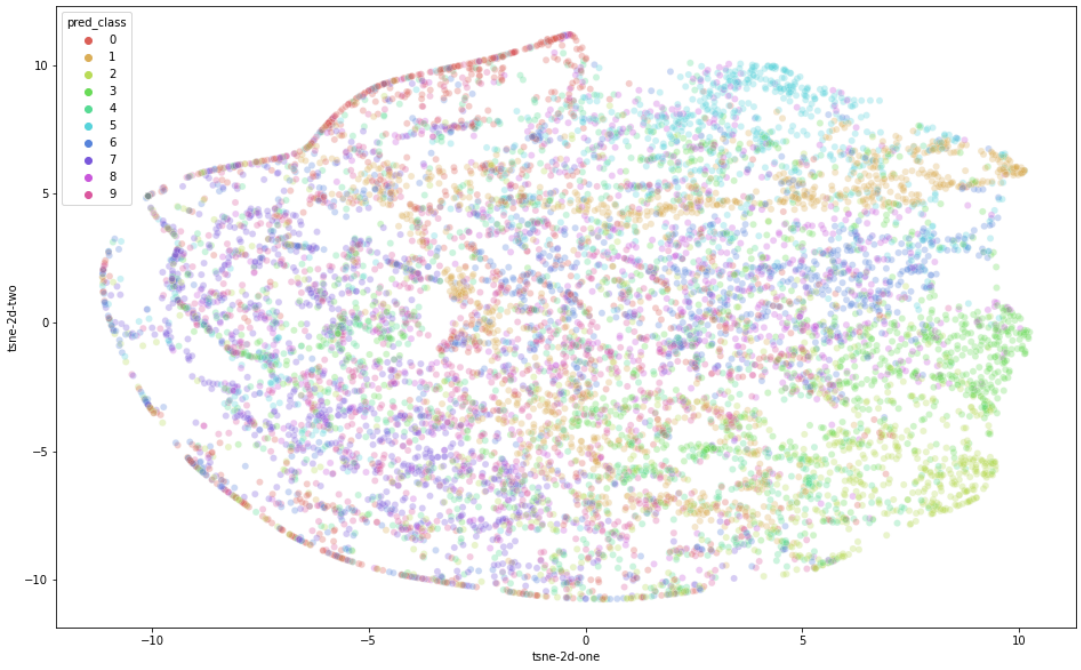
```
(10000, 5)
```

```
tsne = TSNE(n_components=2, verbose=1, perplexity=40, n_iter=300)
tsne_results = tsne.fit_transform(data_subset)
```

```
[t-SNE] Computing 121 nearest neighbors...
[t-SNE] Indexed 10000 samples in 0.016s...
[t-SNE] Computed neighbors for 10000 samples in 0.439s...
[t-SNE] Computed conditional probabilities for sample 1000 / 10000
[t-SNE] Computed conditional probabilities for sample 2000 / 10000
[t-SNE] Computed conditional probabilities for sample 3000 / 10000
[t-SNE] Computed conditional probabilities for sample 4000 / 10000
[t-SNE] Computed conditional probabilities for sample 5000 / 10000
[t-SNE] Computed conditional probabilities for sample 6000 / 10000
[t-SNE] Computed conditional probabilities for sample 7000 / 10000
[t-SNE] Computed conditional probabilities for sample 8000 / 10000
[t-SNE] Computed conditional probabilities for sample 9000 / 10000
[t-SNE] Computed conditional probabilities for sample 10000 / 10000
[t-SNE] Mean sigma: 0.000000
[t-SNE] KL divergence after 250 iterations with early exaggeration: 81.876999
[t-SNE] KL divergence after 300 iterations: 2.650776
```

```
activation_df_subset['tsne-2d-one'] = tsne_results[:,0]
activation_df_subset['tsne-2d-two'] = tsne_results[:,1]

plt.figure(figsize=(16,10))
sns.scatterplot(
    x="tsne-2d-one", y="tsne-2d-two",
    hue="pred_class",
    palette=sns.color_palette("hls", 10),
    data=activation_df_subset,
    legend="full",
    alpha=0.3
)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f84bcea7ac8>
```



Reducing dimensionality of the data with Random Forests.

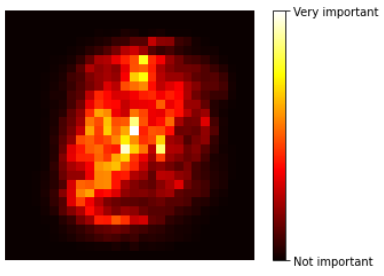
We create a Random Forrest Classifier (with the default 100 trees) and use it to find the relative importance of the 784 features (pixels) in the training set. We produce a heat map to visual the relative importance of the features (using code from Hands On Machine Learning by A. Geron). Finally, we select the 70 most important feature (pixels) from the training, validation and test images to test our 'best' model on.

Correlation matrix that measures the linear relationships
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html

```
rnd_clf = RandomForestClassifier(n_estimators=100, random_state=42)
rnd_clf.fit(x_train_norm,y_train_encoded)
```

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=42, verbose=0,
                       warm_start=False)
```

```
def plot_digit(data):
    image = data.reshape(28, 28)
    plt.imshow(image, cmap = 'hot',
               interpolation="nearest")
    plt.axis("off")

plot_digit(rnd_clf.feature_importances_)
cbar = plt.colorbar(ticks=[rnd_clf.feature_importances_.min(), rnd_clf.feature_importances_.max()])
cbar.ax.set_yticklabels(['Not important', 'Very important'])
plt.show()
```



Correlation matrix that measures the linear relationships
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html

```
# https://stackoverflow.com/questions/6910641/how-do-i-get-indices-of-n-maximum-values-in-a-numpy-array
n = 70
imp_arr = rnd_clf.feature_importances_
idx = (-imp_arr).argsort()[:n]          # get the indices of the 70 "most important" features/pixels
len(idx)
```

```
70
```

```
# Create training and test images using just the 70 pixel locations obtained above
train_images_sm = x_train_norm[:,idx]
test_images_sm = x_test_norm[:,idx]
```

```
train_images_sm.shape, test_images_sm.shape # the reduced images have dimension 70
```

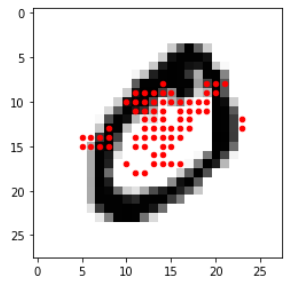
```
((60000, 70), (10000, 70))
```

Visualizing the 70 pixels

We convert the array of indexes to ordered pairs and plot them as red circles on the second training image. These are the features (pixels) we train our neural network on.

```
# to convert an index n, 0<= n < 784
def pair(n,size):
    x = n//size
    y = n%size
    return x,y

plt.imshow(x_train_norm[1].reshape(28,28),cmap='binary')
x, y = np.array([pair(k,28) for k in idx]).T
plt.scatter(x,y,color='red',s=20)
```

```
<matplotlib.collections.PathCollection at 0x7f848a9049e8>
```



Experiment 5

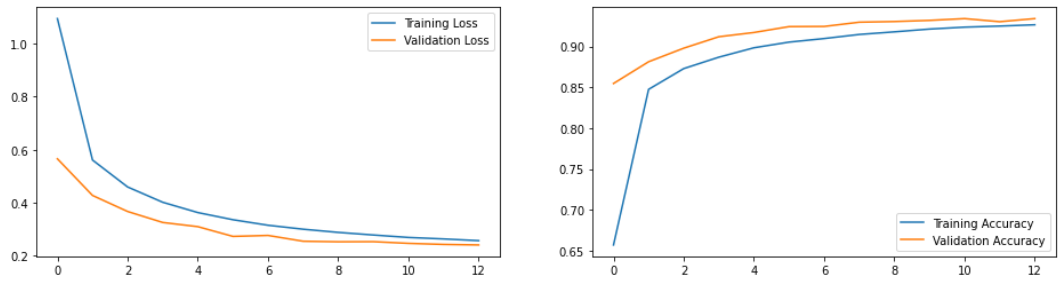
```
model5 = Sequential([
    Dense(input_shape=[70], units = 128, activation = tf.nn.relu),
    Dense(name = "hidden_layer", units = 5, activation = tf.nn.relu),
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])

model5.compile(optimizer='rmsprop',
               loss = 'categorical_crossentropy',
               metrics=['accuracy'])

history = model5.fit(
     train_images_sm
    ,y_train_encoded
    ,epochs = 200
    ,validation_split=0.0833
    ,callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

```
Epoch 1/200
1719/1719 [==============================] - 5s 3ms/step - loss: 1.5060 - accuracy: 0.4876 - val_loss: 0.5655 - val_accuracy: 0.8547
Epoch 2/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.6004 - accuracy: 0.8371 - val_loss: 0.4272 - val_accuracy: 0.8814
Epoch 3/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.4681 - accuracy: 0.8704 - val_loss: 0.3666 - val_accuracy: 0.8980
Epoch 4/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.4067 - accuracy: 0.8854 - val_loss: 0.3252 - val_accuracy: 0.9120
Epoch 5/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3655 - accuracy: 0.8992 - val_loss: 0.3094 - val_accuracy: 0.9172
Epoch 6/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3405 - accuracy: 0.9039 - val_loss: 0.2726 - val_accuracy: 0.9244
Epoch 7/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3177 - accuracy: 0.9089 - val_loss: 0.2763 - val_accuracy: 0.9246
Epoch 8/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.2971 - accuracy: 0.9159 - val_loss: 0.2544 - val_accuracy: 0.9298
Epoch 9/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.2879 - accuracy: 0.9174 - val_loss: 0.2524 - val_accuracy: 0.9306
Epoch 10/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.2738 - accuracy: 0.9231 - val_loss: 0.2528 - val_accuracy: 0.9320
Epoch 11/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.2647 - accuracy: 0.9236 - val_loss: 0.2464 - val_accuracy: 0.9342
Epoch 12/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.2604 - accuracy: 0.9264 - val_loss: 0.2424 - val_accuracy: 0.9304
Epoch 13/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.2537 - accuracy: 0.9265 - val_loss: 0.2403 - val_accuracy: 0.9342
```

```
loss, accuracy = model5.evaluate(test_images_sm, y_test_encoded)
print('test set accuracy: ', accuracy * 100)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.2948 - accuracy: 0.9204
test set accuracy:  92.04000234603882
```

```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
val_accs = history.history['val_accuracy']
epochs = len(losses)

plt.figure(figsize=(16, 4))
for i, metrics in enumerate(zip([losses, accs], [val_losses, val_accs], ['Loss', 'Accuracy'])):
    plt.subplot(1, 2, i + 1)
    plt.plot(range(epochs), metrics[0], label='Training {}'.format(metrics[2]))
    plt.plot(range(epochs), metrics[1], label='Validation {}'.format(metrics[2]))
    plt.legend()
plt.show()
```




```
pred_classes = np.argmax(model5.predict(train_images_sm), axis=-1)
pred_classes
```

```
array([5, 0, 4, ..., 5, 6, 8])
```

```
conf_mx = tf.math.confusion_matrix(y_train, pred_classes)
conf_mx
```

```
<tf.Tensor: shape=(10, 10), dtype=int32, numpy=
```

```
array([[5632,    8,   92,    6,    2,   34,   99,   23,   18,    9],
       [   5, 6603,   23,    4,    4,   18,    5,   27,   52,    1],
       [  32,   60, 5487,   91,   21,   53,   26,  101,   63,   24],
       [   7,    4,  121, 5582,    7,  269,    0,   61,   36,   44],
       [   3,    3,   35,    6, 5347,   63,   80,   16,   53,  236],
       [  30,   19,   23,  138,   46, 5033,   35,   14,   56,   27],
       [  44,   13,   51,    1,   64,   74, 5602,    2,   59,    8],
       [   5,   32,   96,   69,   16,    8,    2, 5900,   17,  120],
       [   2,   50,   85,   54,   50,  116,   29,   18, 5376,   71],
       [   6,    3,   49,   72,  160,   56,   21,   98,   45, 5439]], dtype=int32)>
```

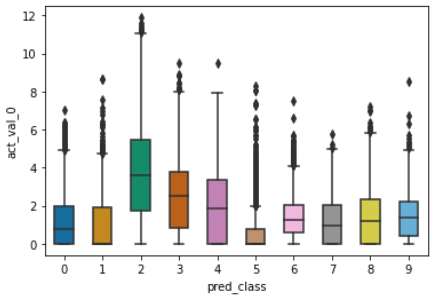
```
# Extracts the outputs of the 2 layers:
layer_outputs = [layer.output for layer in model.layers]

# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

print(f"There are {len(layer_outputs)} layers")
layer_outputs # description of the layers

# Get the outputs of all the hidden nodes for each of the 60000 training images
activations = activation_model.predict(x_train_norm)
hidden_layer_activation = activations[0]
output_layer_activations = activations[1]
hidden_layer_activation.shape   #  each of the 128 hidden nodes has one activation value per training image

output_layer_activations.shape

print(f"The maximum activation value of the hidden nodes in the hidden layer is \
{hidden_layer_activation.max()}")

# Some stats about the output layer as an aside...
np.set_printoptions(suppress = True)  # display probabilities as decimals and NOT in scientific notation
ouput_layer_activation = activations[1]
print(f"The output node has shape {ouput_layer_activation.shape}")
print(f"The output for the first image are {ouput_layer_activation[0].round(4)}")
print(f"The sum of the probabilities is (approximately) {ouput_layer_activation[0].sum()}")
```

```
There are 2 layers
The maximum activation value of the hidden nodes in the hidden layer is 17.98011589050293
The output node has shape (60000, 10)
The output for the first image are [0.    0.    0.    0.002 0.    0.998 0.    0.    0.    0.   ]
The sum of the probabilities is (approximately) 1.0
```

```
#Get the dataframe of all the node values
activation_data = {'pred_class':pred_classes}
for k in range(0,5):
    activation_data[f"act_val_{k}"] = hidden_layer_activation[:,k]

activation_df = pd.DataFrame(activation_data)
activation_df.head()
```

	pred_class	act_val_0	act_val_1	act_val_2	act_val_3	act_val_4
0	5	0.711956	1.275676	0.507729	0.361824	2.179609
1	0	1.117060	1.126043	0.000000	0.000000	0.000000
2	4	1.001708	1.550730	2.589451	1.804716	0.421158
3	1	3.044749	0.790666	1.341971	0.000000	0.000000
4	9	2.145078	1.221523	3.052742	0.000000	0.000000

```
# To see how closely the hidden node activation values correlate with the class predictions
# Let us use seaborn for the boxplots this time.
bplot = sns.boxplot(y='act_val_0', x='pred_class',
                    data=activation_df[['act_val_0','pred_class']],
                    width=0.5,
                    palette="colorblind")
```

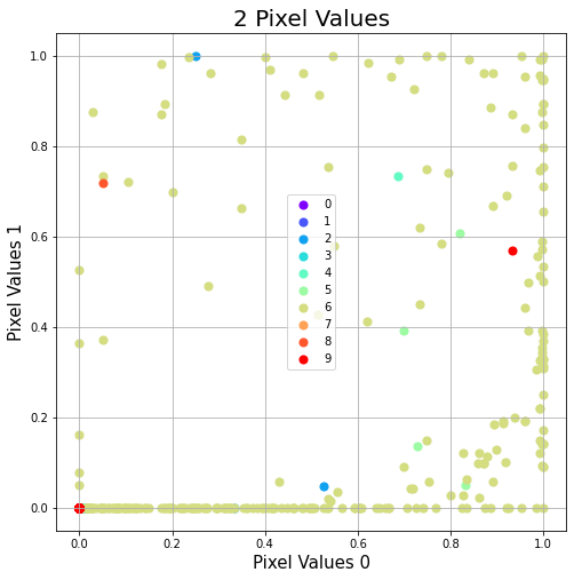



```
pixel_data = {'pred_class':pred_classes}
for k in range(0,784):
    pixel_data[f"pix_val_{k}"] = x_train_norm[:,k]
pixel_df = pd.DataFrame(pixel_data)
pixel_df.head()
```

	pred_class	pix_val_0	pix_val_1	pix_val_2	pix_val_3	pix_val_4	pix_val_5	pix_val_6	pix_val_7	pix_val_8	pix_val_9	pix_val_10	pix_val_11	pix_val_12	pix_val_13	pix_val_14	pix_val_15	pix_val_16	pix_val_17	pix_val_1
0	5	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
1	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
2	4	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
3	1	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
4	9	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

5 rows × 785 columns

```
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Pixel Values 0', fontsize = 15)
ax.set_ylabel('Pixel Values 1', fontsize = 15)
ax.set_title('2 Pixel Values', fontsize = 20)
targets = [0,1,2,3,4,5,6,7,8,9]
#colors = ['r', 'g', 'b']
from matplotlib import cm


colors = cm.rainbow(np.linspace(0, 1, 10))

for target, color in zip(targets,colors):
    indicesToKeep = pixel_df['pred_class'] == target
    ax.scatter(pixel_df.loc[indicesToKeep, 'pix_val_77']
               , pixel_df.loc[indicesToKeep, 'pix_val_78']
               , c = color.reshape(1,-1)
               , s = 50)
ax.legend(targets)
ax.grid()
```



Experiment 6

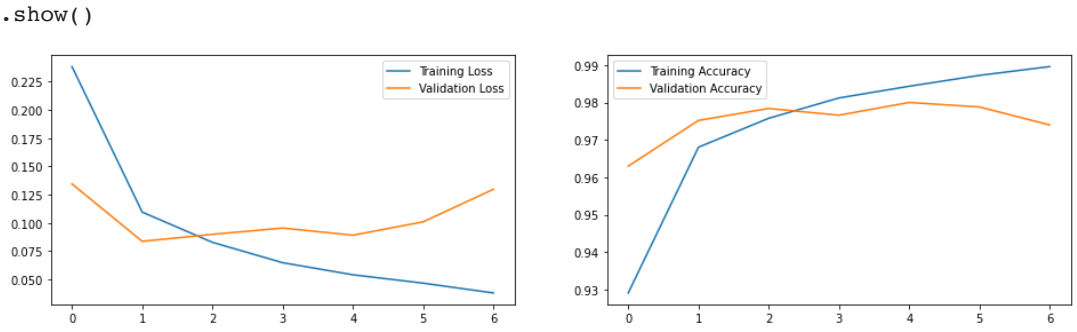
```
model6 = Sequential([
    Dense(input_shape=[784], units = 128, activation = tf.nn.relu),
    Dense(name = "hidden_layer", units = 128, activation = tf.nn.relu),
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])

model6.compile(optimizer='rmsprop',
               loss = 'categorical_crossentropy',
               metrics=['accuracy'])

history = model6.fit(
    x_train_norm
    ,y_train_encoded
    ,epochs = 200
    ,validation_split=0.0833
    ,callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

```
Epoch 1/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3895 - accuracy: 0.8856 - val_loss: 0.1345 - val_accuracy: 0.9630
Epoch 2/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1090 - accuracy: 0.9678 - val_loss: 0.0837 - val_accuracy: 0.9752
Epoch 3/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0805 - accuracy: 0.9763 - val_loss: 0.0899 - val_accuracy: 0.9784
Epoch 4/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0577 - accuracy: 0.9830 - val_loss: 0.0955 - val_accuracy: 0.9766
Epoch 5/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0516 - accuracy: 0.9853 - val_loss: 0.0891 - val_accuracy: 0.9800
Epoch 6/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0440 - accuracy: 0.9883 - val_loss: 0.1009 - val_accuracy: 0.9788
Epoch 7/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0356 - accuracy: 0.9904 - val_loss: 0.1297 - val_accuracy: 0.9740
```

```
loss, accuracy = model6.evaluate(x_test_norm, y_test_encoded)
print('test set accuracy: ', accuracy * 100)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.1249 - accuracy: 0.9738
test set accuracy:  97.3800003528595
```

```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
val_accs = history.history['val_accuracy']
epochs = len(losses)

plt.figure(figsize=(16, 4))
for i, metrics in enumerate(zip([losses, accs], [val_losses, val_accs], ['Loss', 'Accuracy'])):
    plt.subplot(1, 2, i + 1)
    plt.plot(range(epochs), metrics[0], label='Training {}'.format(metrics[2]))
    plt.plot(range(epochs), metrics[1], label='Validation {}'.format(metrics[2]))
    plt.legend()
plt.show()
```




```
pred_classes = np.argmax(model6.predict(x_train_norm), axis=-1)
pred_classes
```

```
array([5, 0, 4, ..., 5, 6, 8])
```

```
conf_mx = tf.math.confusion_matrix(y_train, pred_classes)
conf_mx
```

```
<tf.Tensor: shape=(10, 10), dtype=int32, numpy=
array([[5914,    0,    2,    0,    0,    1,    0,    1,    2,    3],
       [   0, 6696,   12,    3,    1,    0,    0,   13,   15,    2],
       [  10,    1, 5936,    0,    0,    0,    0,    5,    4,    2],
       [   6,    1,   52, 5984,    0,   28,    0,   15,   24,   21],
       [   7,    7,    1,    0, 5688,    0,    5,    4,   11,  119],
       [   5,    1,    5,   15,    0, 5339,   13,    0,   28,   15],
       [  46,    3,    5,    0,    3,    6, 5847,    0,    7,    1],
       [   1,    4,   14,    4,    2,    1,    0, 6194,    4,   41],
       [  29,    3,   14,    6,    1,    3,    0,    4, 5778,   13],
       [   7,    0,    3,    4,    3,    4,    0,    6,   18, 5904]], dtype=int32)>
```

```
pred_classes2 = np.argmax(model6.predict(x_test_norm), axis=-1)
pred_classes2
```

```
array([7, 2, 1, ..., 4, 5, 6])
```

```
conf_mx2 = tf.math.confusion_matrix(y_test, pred_classes2)
conf_mx2
```

```
<tf.Tensor: shape=(10, 10), dtype=int32, numpy=
array([[ 972,    0,    0,    2,    0,    0,    1,    1,    3,    1],
       [   0, 1122,    4,    0,    0,    0,    2,    2,    5,    0],
       [   3,    1, 1016,    0,    2,    0,    0,    5,    4,    1],
```

```
[   0,    0,   11,  980,    0,    7,    0,    5,    1,    6],
[   4,    0,    2,    1,  941,    0,    7,    4,    1,   22],
[   2,    0,    1,    9,    1,  864,    5,    1,    6,    3],
[  12,    2,    2,    1,    5,    4,  930,    0,    2,    0],
[   1,    1,   11,    1,    0,    0,    0,  998,    3,   13],
[  14,    0,    8,    6,    3,    5,    2,    5,  929,    2],
[   2,    2,    1,    2,    2,    2,    0,    3,    9,  986]], dtype=int32)>
```

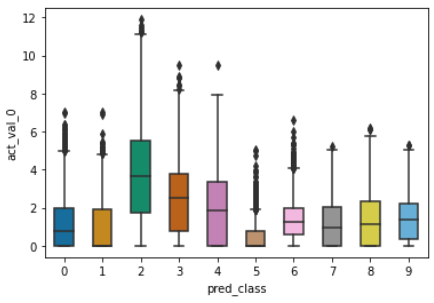
```
# Extracts the outputs of the 2 layers:
layer_outputs = [layer.output for layer in model.layers]

# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

print(f"There are {len(layer_outputs)} layers")
layer_outputs # description of the layers

# Get the outputs of all the hidden nodes for each of the 60000 training images
activations = activation_model.predict(x_train_norm)
hidden_layer_activation = activations[0]
output_layer_activations = activations[1]
hidden_layer_activation.shape   #  each of the 128 hidden nodes has one activation value per training image

output_layer_activations.shape

print(f"The maximum activation value of the hidden nodes in the hidden layer is \
{hidden_layer_activation.max()}")

# Some stats about the output layer as an aside...
np.set_printoptions(suppress = True)  # display probabilities as decimals and NOT in scientific notation
ouput_layer_activation = activations[1]
print(f"The output node has shape {ouput_layer_activation.shape}")
print(f"The output for the first image are {ouput_layer_activation[0].round(4)}")
print(f"The sum of the probabilities is (approximately) {ouput_layer_activation[0].sum()}")
```

```
There are 2 layers
The maximum activation value of the hidden nodes in the hidden layer is 17.98011589050293
The output node has shape (60000, 10)
The output for the first image are [0.    0.    0.    0.002 0.    0.998 0.    0.    0.    0.   ]
The sum of the probabilities is (approximately) 1.0
```

```
#Get the dataframe of all the node values
activation_data = {'pred_class':pred_classes}
for k in range(0,5):
    activation_data[f"act_val_{k}"] = hidden_layer_activation[:,k]

activation_df = pd.DataFrame(activation_data)
activation_df.head()
```

	pred_class	act_val_0	act_val_1	act_val_2	act_val_3	act_val_4
0	5	0.711956	1.275676	0.507729	0.361824	2.179609
1	0	1.117060	1.126043	0.000000	0.000000	0.000000
2	4	1.001708	1.550730	2.589451	1.804716	0.421158
3	1	3.044749	0.790666	1.341971	0.000000	0.000000
4	9	2.145078	1.221523	3.052742	0.000000	0.000000

```
# To see how closely the hidden node activation values correlate with the class predictions
# Let us use seaborn for the boxplots this time.
bplot = sns.boxplot(y='act_val_0', x='pred_class',
                 data=activation_df[['act_val_0','pred_class']],
                 width=0.5,
                 palette="colorblind")
```

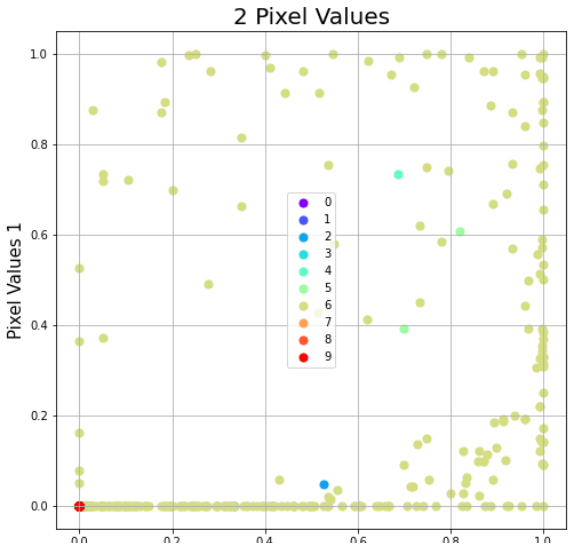



```
pixel_data = {'pred_class':pred_classes}
for k in range(0,784):
    pixel_data[f"pix_val_{k}"] = x_train_norm[:,k]
pixel_df = pd.DataFrame(pixel_data)
pixel_df.head()
```

	pred_class	pix_val_0	pix_val_1	pix_val_2	pix_val_3	pix_val_4	pix_val_5	pix_val_6	pix_val_7	pix_val_8	pix_val_9	pix_val_10	pix_val_11	pix_val_12	pix_val_13	pix_val_14	pix_val_15	pix_val_16	pix_val_17	pix_val_1
0	5	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
1	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
2	4	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
3	1	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
4	9	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

5 rows × 785 columns

```
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Pixel Values 0', fontsize = 15)
ax.set_ylabel('Pixel Values 1', fontsize = 15)
ax.set_title('2 Pixel Values', fontsize = 20)
targets = [0,1,2,3,4,5,6,7,8,9]
#colors = ['r', 'g', 'b']
from matplotlib import cm


colors = cm.rainbow(np.linspace(0, 1, 10))

for target, color in zip(targets,colors):
    indicesToKeep = pixel_df['pred_class'] == target
    ax.scatter(pixel_df.loc[indicesToKeep, 'pix_val_77']
               , pixel_df.loc[indicesToKeep, 'pix_val_78']
               , c = color.reshape(1,-1)
               , s = 50)
ax.legend(targets)
ax.grid()
```



Experiment 7

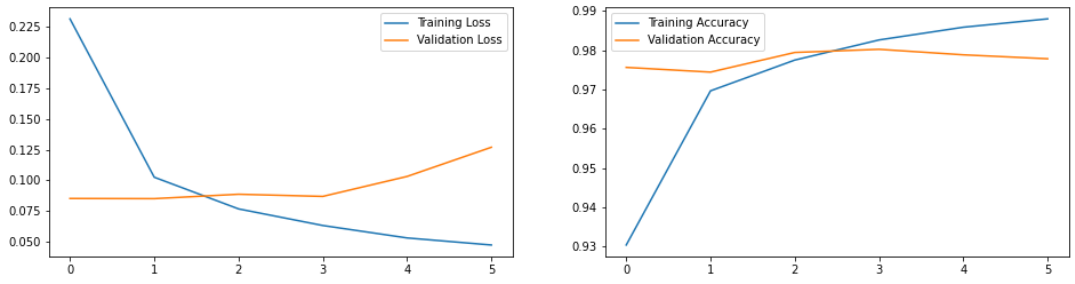
```
model7 = Sequential([
    Dense(input_shape=[784], units = 128, activation = tf.nn.relu),
    Dense(name = "hidden_layer", units = 256, activation = tf.nn.relu),
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])

model7.compile(optimizer='rmsprop',
               loss = 'categorical_crossentropy',
               metrics=['accuracy'])

history = model7.fit(
    x_train_norm
    ,y_train_encoded
    ,epochs = 200
    ,validation_split=0.0833
    ,callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

```
Epoch 1/200
1719/1719 [==============================] - 6s 3ms/step - loss: 0.3966 - accuracy: 0.8798 - val_loss: 0.0853 - val_accuracy: 0.9756
Epoch 2/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1048 - accuracy: 0.9682 - val_loss: 0.0852 - val_accuracy: 0.9744
Epoch 3/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0745 - accuracy: 0.9780 - val_loss: 0.0888 - val_accuracy: 0.9794
Epoch 4/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0588 - accuracy: 0.9838 - val_loss: 0.0870 - val_accuracy: 0.9802
Epoch 5/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0516 - accuracy: 0.9858 - val_loss: 0.1032 - val_accuracy: 0.9788
Epoch 6/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0432 - accuracy: 0.9891 - val_loss: 0.1269 - val_accuracy: 0.9778
```

```
loss, accuracy = model7.evaluate(x_test_norm, y_test_encoded)
print('test set accuracy: ', accuracy * 100)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.1272 - accuracy: 0.9737
test set accuracy:  97.36999869346619
```

```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
val_accs = history.history['val_accuracy']
epochs = len(losses)

plt.figure(figsize=(16, 4))
for i, metrics in enumerate(zip([losses, accs], [val_losses, val_accs], ['Loss', 'Accuracy'])):
    plt.subplot(1, 2, i + 1)
    plt.plot(range(epochs), metrics[0], label='Training {}'.format(metrics[2]))
    plt.plot(range(epochs), metrics[1], label='Validation {}'.format(metrics[2]))
    plt.legend()
plt.show()
```




```
pred_classes = np.argmax(model7.predict(x_train_norm), axis=-1)
pred_classes
```

```
array([5, 0, 4, ..., 5, 6, 8])
```

```
conf_mx = tf.math.confusion_matrix(y_train, pred_classes)
conf_mx
```

```
<tf.Tensor: shape=(10, 10), dtype=int32, numpy=
array([[5886,    0,    6,    3,    0,    3,   17,    0,    5,    3],
       [   0, 6731,    3,    1,    1,    0,    3,    0,    3,    0],
       [   3,    5, 5937,    5,    2,    0,    2,    2,    2,    0],
       [   0,    2,   30, 6062,    0,   25,    2,    3,    6,    1],
       [   1,    7,   10,    0, 5797,    3,   14,    1,    5,    4],
       [   1,    3,    2,    8,    0, 5383,   20,    0,    3,    1],
       [   2,    1,    2,    0,    0,    4, 5909,    0,    0,    0],
       [   4,   24,   75,   13,   13,    8,    0, 6110,    8,   10],
       [   3,   12,    8,   45,    1,   22,   31,    0, 5723,    6],
       [   5,    7,    5,   28,   71,   60,    0,   11,   23, 5739]], dtype=int32)>
```

```
# Extracts the outputs of the 2 layers:
layer_outputs = [layer.output for layer in model.layers]

# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

print(f"There are {len(layer_outputs)} layers")
layer_outputs # description of the layers

# Get the outputs of all the hidden nodes for each of the 60000 training images
activations = activation_model.predict(x_train_norm)
hidden_layer_activation = activations[0]
output_layer_activations = activations[1]
hidden_layer_activation.shape   #  each of the 128 hidden nodes has one activation value per training image
```

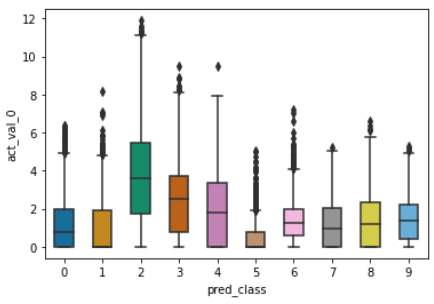
```
output_layer_activations.shape

print(f"The maximum activation value of the hidden nodes in the hidden layer is \
{hidden_layer_activation.max()}")

# Some stats about the output layer as an aside...
np.set_printoptions(suppress = True)  # display probabilities as decimals and NOT in scientific notation
ouput_layer_activation = activations[1]
print(f"The output node has shape {ouput_layer_activation.shape}")
print(f"The output for the first image are {ouput_layer_activation[0].round(4)}")
print(f"The sum of the probabilities is (approximately) {ouput_layer_activation[0].sum()}")
```

```
There are 2 layers
The maximum activation value of the hidden nodes in the hidden layer is 17.98011589050293
The output node has shape (60000, 10)
The output for the first image are [0.    0.    0.    0.002 0.    0.998 0.    0.    0.    0.   ]
The sum of the probabilities is (approximately) 1.0
```

```
#Get the dataframe of all the node values
activation_data = {'pred_class':pred_classes}
for k in range(0,5):
    activation_data[f"act_val_{k}"] = hidden_layer_activation[:,k]

activation_df = pd.DataFrame(activation_data)
activation_df.head()
```

	pred_class	act_val_0	act_val_1	act_val_2	act_val_3	act_val_4
0	5	0.711956	1.275676	0.507729	0.361824	2.179609
1	0	1.117060	1.126043	0.000000	0.000000	0.000000
2	4	1.001708	1.550730	2.589451	1.804716	0.421158
3	1	3.044749	0.790666	1.341971	0.000000	0.000000
4	9	2.145078	1.221523	3.052742	0.000000	0.000000

```
# To see how closely the hidden node activation values correlate with the class predictions
# Let us use seaborn for the boxplots this time.
bplot = sns.boxplot(y='act_val_0', x='pred_class',
                    data=activation_df[['act_val_0','pred_class']],
                    width=0.5,
                    palette="colorblind")
```

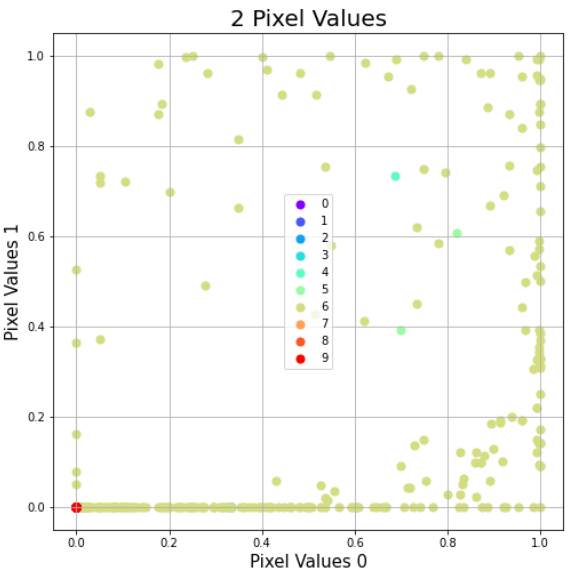



```
pixel_data = {'pred_class':pred_classes}
for k in range(0,784):
    pixel_data[f"pix_val_{k}"] = x_train_norm[:,k]
pixel_df = pd.DataFrame(pixel_data)
pixel_df.head()
```

	pred_class	pix_val_0	pix_val_1	pix_val_2	pix_val_3	pix_val_4	pix_val_5	pix_val_6	pix_val_7	pix_val_8	pix_val_9	pix_val_10	pix_val_11	pix_val_12	pix_val_13	pix_val_14	pix_val_15	pix_val_16	pix_val_17	pix_val_1
0	5	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
1	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
2	4	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
3	1	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
4	9	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

5 rows × 785 columns

```
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Pixel Values 0', fontsize = 15)
ax.set_ylabel('Pixel Values 1', fontsize = 15)
ax.set_title('2 Pixel Values', fontsize = 20)
targets = [0,1,2,3,4,5,6,7,8,9]
#colors = ['r', 'g', 'b']
from matplotlib import cm


colors = cm.rainbow(np.linspace(0, 1, 10))

for target, color in zip(targets,colors):
    indicesToKeep = pixel_df['pred_class'] == target
    ax.scatter(pixel_df.loc[indicesToKeep, 'pix_val_77']
               , pixel_df.loc[indicesToKeep, 'pix_val_78']
               , c = color.reshape(1,-1)
               , s = 50)
ax.legend(targets)
ax.grid()
```



Experiment 8

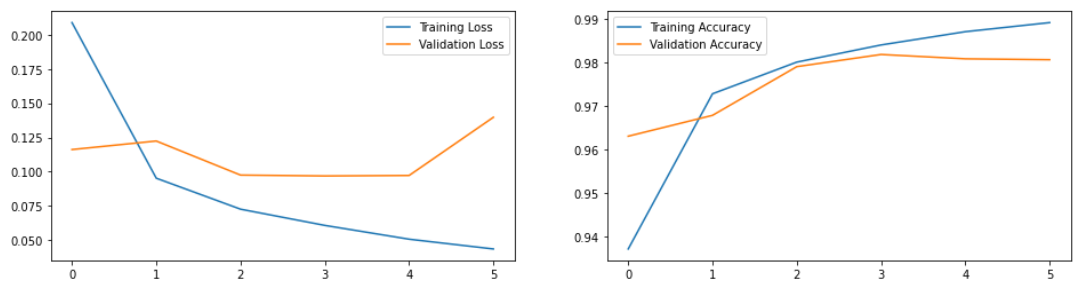
```
model8 = Sequential([
    Dense(input_shape=[784], units = 256, activation = tf.nn.relu),
    Dense(name = "hidden_layer", units = 256, activation = tf.nn.relu),
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])

model8.compile(optimizer='rmsprop',
               loss = 'categorical_crossentropy',
               metrics=['accuracy'])

history = model8.fit(
    x_train_norm
    ,y_train_encoded
    ,epochs = 200
    ,validation_split=0.0833
    ,callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

```
Epoch 1/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3474 - accuracy: 0.8941 - val_loss: 0.1162 - val_accuracy: 0.9630
Epoch 2/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0936 - accuracy: 0.9729 - val_loss: 0.1224 - val_accuracy: 0.9678
Epoch 3/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0694 - accuracy: 0.9807 - val_loss: 0.0975 - val_accuracy: 0.9790
Epoch 4/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0554 - accuracy: 0.9848 - val_loss: 0.0969 - val_accuracy: 0.9818
Epoch 5/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0461 - accuracy: 0.9876 - val_loss: 0.0972 - val_accuracy: 0.9808
Epoch 6/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0409 - accuracy: 0.9896 - val_loss: 0.1397 - val_accuracy: 0.9806
```

```
loss, accuracy = model8.evaluate(x_test_norm, y_test_encoded)
print('test set accuracy: ', accuracy * 100)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.1315 - accuracy: 0.9773
test set accuracy:  97.72999882698059
```

```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
val_accs = history.history['val_accuracy']
epochs = len(losses)

plt.figure(figsize=(16, 4))
for i, metrics in enumerate(zip([losses, accs], [val_losses, val_accs], ['Loss', 'Accuracy'])):
    plt.subplot(1, 2, i + 1)
    plt.plot(range(epochs), metrics[0], label='Training {}'.format(metrics[2]))
    plt.plot(range(epochs), metrics[1], label='Validation {}'.format(metrics[2]))
    plt.legend()
plt.show()
```




```
pred_classes = np.argmax(model8.predict(x_train_norm), axis=-1)
pred_classes
```

```
array([5, 0, 4, ..., 5, 6, 8])
```

```
conf_mx = tf.math.confusion_matrix(y_train, pred_classes)
conf_mx
```

```
<tf.Tensor: shape=(10, 10), dtype=int32, numpy=
array([[5908,    1,    2,    0,    0,    2,    4,    0,    4,    2],
       [   0, 6732,    2,    3,    1,    0,    1,    2,    1,    0],
       [   3,   11, 5925,    7,    3,    1,    1,    0,    7,    0],
       [   3,    5,   14, 6075,    0,    3,    0,    7,   20,    4],
       [   1,   15,    2,    0, 5805,    0,    4,    5,    1,    9],
       [   3,    1,    3,   20,    1, 5347,   32,    0,   12,    2],
       [   3,    1,    4,    0,    2,    0, 5907,    0,    1,    0],
       [   7,   24,   31,    4,    0,    0,    0, 6194,    1,    4],
       [   7,   23,    3,    8,    0,    2,    7,    4, 5797,    0],
       [  13,   15,    2,   12,   41,    8,    1,   40,   30, 5787]], dtype=int32)>
```

```
# Extracts the outputs of the 2 layers:
layer_outputs = [layer.output for layer in model.layers]

# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

print(f"There are {len(layer_outputs)} layers")
layer_outputs # description of the layers

# Get the outputs of all the hidden nodes for each of the 60000 training images
activations = activation_model.predict(x_train_norm)
hidden_layer_activation = activations[0]
output_layer_activations = activations[1]
hidden_layer_activation.shape   #  each of the 128 hidden nodes has one activation value per training image

output_layer_activations.shape

print(f"The maximum activation value of the hidden nodes in the hidden layer is \
{hidden_layer_activation.max()}")

# Some stats about the output layer as an aside...
np.set_printoptions(suppress = True)  # display probabilities as decimals and NOT in scientific notation
ouput_layer_activation = activations[1]
print(f"The output node has shape {ouput_layer_activation.shape}")
print(f"The output for the first image are {ouput_layer_activation[0].round(4)}")
print(f"The sum of the probabilities is (approximately) {ouput_layer_activation[0].sum()}")
```

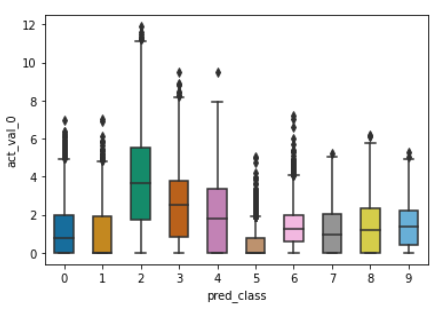
```
There are 2 layers
The maximum activation value of the hidden nodes in the hidden layer is 17.98011589050293
The output node has shape (60000, 10)
The output for the first image are [0.    0.    0.    0.002 0.    0.998 0.    0.    0.    0.   ]
The sum of the probabilities is (approximately) 1.0
```

```
#Get the dataframe of all the node values
activation_data = {'pred_class':pred_classes}
```

```
activation_data = {'pred_class':pred_classes}
for k in range(0,5):
    activation_data[f"act_val_{k}"] = hidden_layer_activation[:,k]

activation_df = pd.DataFrame(activation_data)
activation_df.head()
```

	pred_class	act_val_0	act_val_1	act_val_2	act_val_3	act_val_4
0	5	0.711956	1.275676	0.507729	0.361824	2.179609
1	0	1.117060	1.126043	0.000000	0.000000	0.000000
2	4	1.001708	1.550730	2.589451	1.804716	0.421158
3	1	3.044749	0.790666	1.341971	0.000000	0.000000
4	9	2.145078	1.221523	3.052742	0.000000	0.000000

```
# To see how closely the hidden node activation values correlate with the class predictions
# Let us use seaborn for the boxplots this time.
bplot = sns.boxplot(y='act_val_0', x='pred_class',
                    data=activation_df[['act_val_0','pred_class']],
                    width=0.5,
                    palette="colorblind")
```

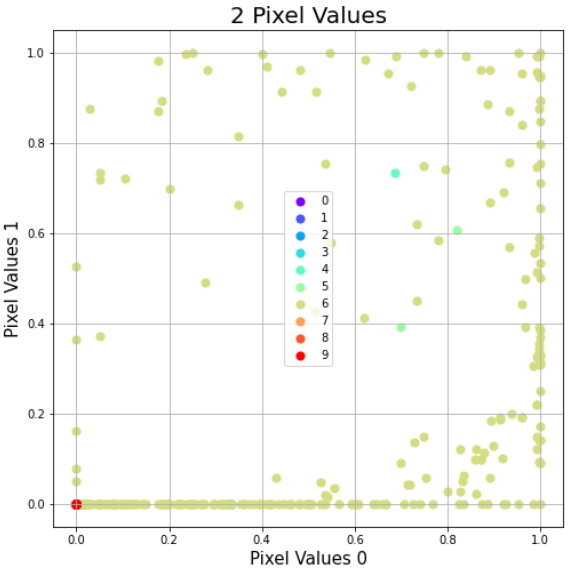



```
pixel_data = {'pred_class':pred_classes}
for k in range(0,784):
    pixel_data[f"pix_val_{k}"] = x_train_norm[:,k]
pixel_df = pd.DataFrame(pixel_data)
pixel_df.head()
```

	pred_class	pix_val_0	pix_val_1	pix_val_2	pix_val_3	pix_val_4	pix_val_5	pix_val_6	pix_val_7	pix_val_8	pix_val_9	pix_val_10	pix_val_11	pix_val_12	pix_val_13	pix_val_14	pix_val_15	pix_val_16	pix_val_17	pix_val_1
0	5	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
1	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
2	4	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
3	1	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
4	9	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

5 rows × 785 columns

```
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Pixel Values 0', fontsize = 15)
ax.set_ylabel('Pixel Values 1', fontsize = 15)
ax.set_title('2 Pixel Values', fontsize = 20)
targets = [0,1,2,3,4,5,6,7,8,9]
#colors = ['r', 'g', 'b']
from matplotlib import cm


colors = cm.rainbow(np.linspace(0, 1, 10))

for target, color in zip(targets,colors):
    indicesToKeep = pixel_df['pred_class'] == target
    ax.scatter(pixel_df.loc[indicesToKeep, 'pix_val_77']
               , pixel_df.loc[indicesToKeep, 'pix_val_78']
               , c = color.reshape(1,-1)
               , s = 50)
ax.legend(targets)
ax.grid()
```



Experiment 9

```
model9 = Sequential([
    Dense(input_shape=[784], units = 512, activation = tf.nn.relu),
    Dense(name = "hidden_layer", units = 512, activation = tf.nn.relu),
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])

model9.compile(optimizer='rmsprop',
               loss = 'categorical_crossentropy',
               metrics=['accuracy'])

history = model9.fit(
    x_train_norm
    ,y_train_encoded
    ,epochs = 200
    ,validation_split=0.0833
```

```
    ,callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

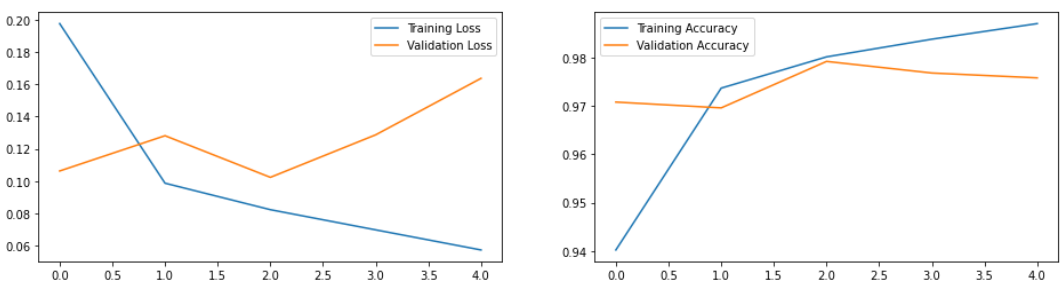
```
Epoch 1/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3231 - accuracy: 0.8972 - val_loss: 0.1063 - val_accuracy: 0.9708
Epoch 2/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0960 - accuracy: 0.9738 - val_loss: 0.1282 - val_accuracy: 0.9696
Epoch 3/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0763 - accuracy: 0.9805 - val_loss: 0.1024 - val_accuracy: 0.9792
Epoch 4/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0607 - accuracy: 0.9845 - val_loss: 0.1287 - val_accuracy: 0.9768
Epoch 5/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0530 - accuracy: 0.9875 - val_loss: 0.1637 - val_accuracy: 0.9758
```

```
loss, accuracy = model9.evaluate(x_test_norm, y_test_encoded)
print('test set accuracy: ', accuracy * 100)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.1442 - accuracy: 0.9769
test set accuracy:  97.68999814987183
```

```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
val_accs = history.history['val_accuracy']
epochs = len(losses)
```

```
plt.figure(figsize=(16, 4))
for i, metrics in enumerate(zip([losses, accs], [val_losses, val_accs], ['Loss', 'Accuracy'])):
    plt.subplot(1, 2, i + 1)
    plt.plot(range(epochs), metrics[0], label='Training {}'.format(metrics[2]))
    plt.plot(range(epochs), metrics[1], label='Validation {}'.format(metrics[2]))
    plt.legend()
plt.show()
```




```
pred_classes = np.argmax(model9.predict(x_train_norm), axis=-1)
pred_classes
```

```
array([5, 0, 4, ..., 5, 6, 8])
```

```
conf_mx = tf.math.confusion_matrix(y_train, pred_classes)
conf_mx
```

```
<tf.Tensor: shape=(10, 10), dtype=int32, numpy=
array([[5796,    5,   31,    1,   20,    3,    6,    3,   55,    3],
       [   0, 6704,    2,    0,    5,    0,    0,   14,   17,    0],
       [   1,   20, 5914,    3,    8,    0,    0,    2,    9,    1],
       [   0,   14,   27, 6016,    2,   15,    0,   11,   42,    4],
       [   0,    1,    3,    0, 5827,    0,    1,    1,    1,    8],
       [   3,    6,    3,   21,    7, 5326,   12,    2,   28,   13],
       [   5,    8,    3,    1,   23,    9, 5847,    0,   22,    0],
       [   1,    8,    7,    2,    9,    0,    0, 6230,    2,    6],
       [   0,    6,    6,    3,   10,    6,    2,    3, 5811,    4],
       [   2,    2,    1,   11,   81,    9,    0,   34,   42, 5767]], dtype=int32)>
```

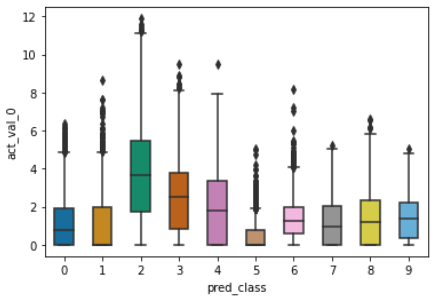
```
# Extracts the outputs of the 2 layers:
layer_outputs = [layer.output for layer in model.layers]

# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

print(f"There are {len(layer_outputs)} layers")
layer_outputs # description of the layers

# Get the outputs of all the hidden nodes for each of the 60000 training images
activations = activation_model.predict(x_train_norm)
hidden_layer_activation = activations[0]
output_layer_activations = activations[1]
hidden_layer_activation.shape   #  each of the 128 hidden nodes has one activation value per training image

output_layer_activations.shape
print(f"The maximum activation value of the hidden nodes in the hidden layer is \
{hidden_layer_activation.max()}")

# Some stats about the output layer as an aside...
np.set_printoptions(suppress = True)  # display probabilities as decimals and NOT in scientific notation
ouput_layer_activation = activations[1]
print(f"The output node has shape {ouput_layer_activation.shape}")
print(f"The output for the first image are {ouput_layer_activation[0].round(4)}")
print(f"The sum of the probabilities is (approximately) {ouput_layer_activation[0].sum()}")
```

```
There are 2 layers
The maximum activation value of the hidden nodes in the hidden layer is 17.98011589050293
The output node has shape (60000, 10)
The output for the first image are [0.    0.    0.    0.002 0.    0.998 0.    0.    0.    0.   ]
The sum of the probabilities is (approximately) 1.0
```

```
#Get the dataframe of all the node values
activation_data = {'pred_class':pred_classes}
for k in range(0,5):
    activation_data[f"act_val_{k}"] = hidden_layer_activation[:,k]

activation_df = pd.DataFrame(activation_data)
activation_df.head()
```

	pred_class	act_val_0	act_val_1	act_val_2	act_val_3	act_val_4
0	5	0.711956	1.275676	0.507729	0.361824	2.179609
1	0	1.117060	1.126043	0.000000	0.000000	0.000000
2	4	1.001708	1.550730	2.589451	1.804716	0.421158
3	1	3.044749	0.790666	1.341971	0.000000	0.000000
4	9	2.145078	1.221523	3.052742	0.000000	0.000000

```
# To see how closely the hidden node activation values correlate with the class predictions
# Let us use seaborn for the boxplots this time.
bplot = sns.boxplot(y='act_val_0', x='pred_class',
                 data=activation_df[['act_val_0','pred_class']],
                 width=0.5,
                 palette="colorblind")
```



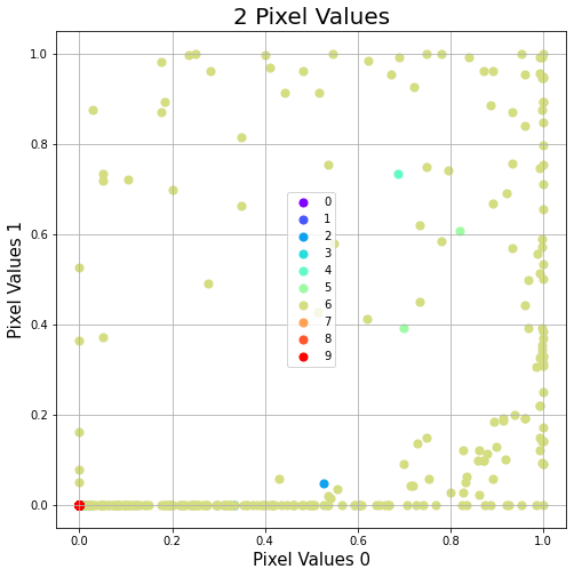

```
pixel_data = {'pred_class':pred_classes}
for k in range(0,784):
    pixel_data[f"pix_val_{k}"] = x_train_norm[:,k]
pixel_df = pd.DataFrame(pixel_data)
pixel_df.head()
```

	pred_class	pix_val_0	pix_val_1	pix_val_2	pix_val_3	pix_val_4	pix_val_5	pix_val_6	pix_val_7	pix_val_8	pix_val_9	pix_val_10	pix_val_11	pix_val_12	pix_val_13	pix_val_14	pix_val_15	pix_val_16	pix_val_17	pix_val_1
0	5	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
1	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
2	4	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
3	1	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
4	9	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

5 rows × 785 columns

```
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Pixel Values 0', fontsize = 15)
ax.set_ylabel('Pixel Values 1', fontsize = 15)
ax.set_title('2 Pixel Values', fontsize = 20)
targets = [0,1,2,3,4,5,6,7,8,9]
#colors = ['r', 'g', 'b']
from matplotlib import cm
```

```
colors = cm.rainbow(np.linspace(0, 1, 10))

for target, color in zip(targets,colors):
    indicesToKeep = pixel_df['pred_class'] == target
    ax.scatter(pixel_df.loc[indicesToKeep, 'pix_val_77']
               , pixel_df.loc[indicesToKeep, 'pix_val_78']
               , c = color.reshape(1,-1)
               , s = 50)
ax.legend(targets)
ax.grid()
```



Experiment 10

```
model10 = Sequential([
    Dense(input_shape=[784], units = 64, activation = tf.nn.relu),
    Dense(name = "hidden_layer", units = 64, activation = tf.nn.relu),
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])

model10.compile(optimizer='rmsprop',
               loss = 'categorical_crossentropy',
               metrics=['accuracy'])

history = model10.fit(
    x_train_norm
    ,y_train_encoded
    ,epochs = 200
    ,validation_split=0.0833
    ,callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

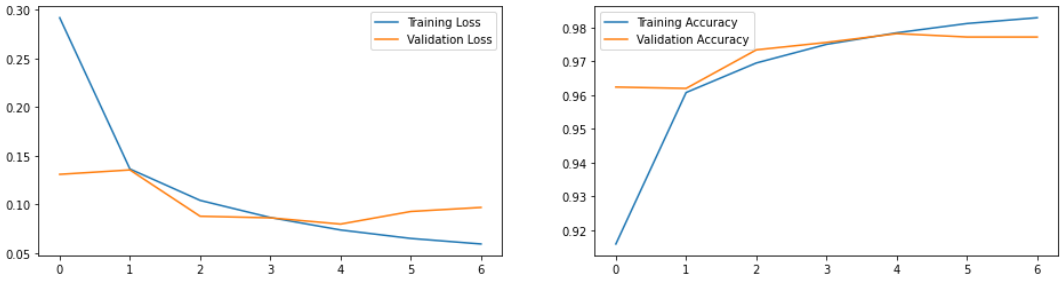
```
Epoch 1/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.4925 - accuracy: 0.8616 - val_loss: 0.1308 - val_accuracy: 0.9624
Epoch 2/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1465 - accuracy: 0.9582 - val_loss: 0.1353 - val_accuracy: 0.9620
Epoch 3/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1019 - accuracy: 0.9693 - val_loss: 0.0877 - val_accuracy: 0.9734
Epoch 4/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0846 - accuracy: 0.9746 - val_loss: 0.0861 - val_accuracy: 0.9756
Epoch 5/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0737 - accuracy: 0.9787 - val_loss: 0.0797 - val_accuracy: 0.9782
Epoch 6/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0635 - accuracy: 0.9814 - val_loss: 0.0926 - val_accuracy: 0.9772
Epoch 7/200
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0566 - accuracy: 0.9828 - val_loss: 0.0968 - val_accuracy: 0.9772
```

```
loss, accuracy = model10.evaluate(x_test_norm, y_test_encoded)
print('test set accuracy: ', accuracy * 100)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.1087 - accuracy: 0.9716
test set accuracy:  97.15999960899353
```

```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
```

```
val_accs = history.history['val_accuracy']
epochs = len(losses)

plt.figure(figsize=(16, 4))
for i, metrics in enumerate(zip([losses, accs], [val_losses, val_accs], ['Loss', 'Accuracy'])):
    plt.subplot(1, 2, i + 1)
    plt.plot(range(epochs), metrics[0], label='Training {}'.format(metrics[2]))
    plt.plot(range(epochs), metrics[1], label='Validation {}'.format(metrics[2]))
    plt.legend()
plt.show()
```




```
pred_classes = np.argmax(model10.predict(x_train_norm), axis=-1)
pred_classes
```

```
array([5, 0, 4, ..., 5, 6, 8])
```

```
conf_mx = tf.math.confusion_matrix(y_train, pred_classes)
conf_mx
```

```
<tf.Tensor: shape=(10, 10), dtype=int32, numpy=
array([[5886,    0,    9,    4,    3,    1,    1,    0,    6,   13],
       [   2, 6690,   14,    4,    1,    1,    0,   17,   11,    2],
       [   5,    7, 5869,   28,    5,    0,    1,   31,    9,    3],
       [   2,    4,   10, 6065,    1,    8,    0,   10,   18,   13],
       [   0,   14,    4,    1, 5763,    2,    2,    6,    2,   48],
       [   7,    7,    3,   58,    5, 5288,   13,    2,   19,   19],
       [  28,    7,    5,    2,   12,   15, 5827,    0,   21,    1],
       [   2,    5,    8,    8,   14,    0,    0, 6222,    1,    5],
       [   6,   13,   10,   31,    2,   10,    6,   11, 5743,   19],
       [   3,    3,    0,    6,   13,    7,    0,   37,   10, 5870]], dtype=int32)>
```

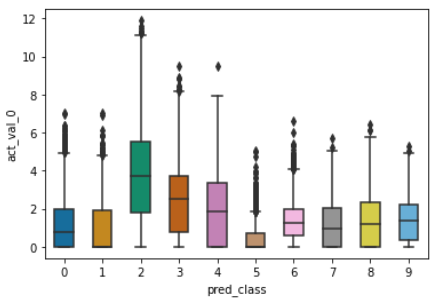
```
# Extracts the outputs of the 2 layers:
layer_outputs = [layer.output for layer in model.layers]

# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

print(f"There are {len(layer_outputs)} layers")
layer_outputs # description of the layers

# Get the outputs of all the hidden nodes for each of the 60000 training images
activations = activation_model.predict(x_train_norm)
hidden_layer_activation = activations[0]
output_layer_activations = activations[1]
hidden_layer_activation.shape   #  each of the 128 hidden nodes has one activation value per training image

output_layer_activations.shape

print(f"The maximum activation value of the hidden nodes in the hidden layer is \
{hidden_layer_activation.max()}")

# Some stats about the output layer as an aside...
np.set_printoptions(suppress = True)  # display probabilities as decimals and NOT in scientific notation
ouput_layer_activation = activations[1]
print(f"The output node has shape {ouput_layer_activation.shape}")
print(f"The output for the first image are {ouput_layer_activation[0].round(4)}")
print(f"The sum of the probabilities is (approximately) {ouput_layer_activation[0].sum()}")
```

```
There are 2 layers
The maximum activation value of the hidden nodes in the hidden layer is 17.98011589050293
The output node has shape (60000, 10)
The output for the first image are [0.    0.    0.    0.002 0.    0.998 0.    0.    0.    0.   ]
The sum of the probabilities is (approximately) 1.0
```

```
#Get the dataframe of all the node values
activation_data = {'pred_class':pred_classes}
for k in range(0,5):
    activation_data[f"act_val_{k}"] = hidden_layer_activation[:,k]

activation_df = pd.DataFrame(activation_data)
activation_df.head()
```

	pred_class	act_val_0	act_val_1	act_val_2	act_val_3	act_val_4
0	5	0.711956	1.275676	0.507729	0.361824	2.179609
1	0	1.117060	1.126043	0.000000	0.000000	0.000000
2	4	1.001708	1.550730	2.589451	1.804716	0.421158
3	1	3.044749	0.790666	1.341971	0.000000	0.000000
4	9	2.145078	1.221523	3.052742	0.000000	0.000000

```
# To see how closely the hidden node activation values correlate with the class predictions
# Let us use seaborn for the boxplots this time.
bplot = sns.boxplot(y='act_val_0', x='pred_class',
                 data=activation_df[['act_val_0','pred_class']],
                 width=0.5,
                 palette="colorblind")
```




```
pred_classes2 = np.argmax(model10.predict(x_test_norm), axis=-1)
pred_classes2
```

```
array([7, 2, 1, ..., 4, 5, 6])
```

```
conf_mx2 = tf.math.confusion_matrix(y_test, pred_classes2)
```

conf_mx2

```
<tf.Tensor: shape=(10, 10), dtype=int32, numpy=
array([[ 967,    0,    1,    2,    1,    1,    2,    1,    3,    2],
       [   0, 1124,    2,    2,    0,    1,    1,    2,    3,    0],
       [   6,    3,  997,    5,    2,    1,    3,    9,    6,    0],
       [   1,    0,    2,  997,    0,    2,    0,    5,    1,    2],
       [   3,    0,    6,    0,  943,    0,    2,    7,    3,   18],
       [   3,    0,    0,   22,    1,  845,    6,    3,    6,    6],
       [   6,    3,    0,    2,    9,    2,  929,    0,    6,    1],
       [   1,    3,   12,    7,    2,    0,    0,  996,    3,    4],
       [   1,    0,    6,    9,    5,    5,    0,    4,  940,    4],
       [   1,    5,    0,    6,    5,    1,    0,    9,    4,  978]], dtype=int32)>
```

```
pixel_data = {'pred_class':pred_classes}
for k in range(0,784):
    pixel_data[f"pix_val_{k}"] = x_train_norm[:,k]
pixel_df = pd.DataFrame(pixel_data)
pixel_df.head()
```

	pred_class	pix_val_0	pix_val_1	pix_val_2	pix_val_3	pix_val_4	pix_val_5	pix_val_6	pix_val_7	pix_val_8	pix_val_9	pix_val_10	pix_val_11	pix_val_12	pix_val_13	pix_val_14	pix_val_15	pix_val_16	pix_val_17	pix_val_1
0	5	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
1	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
2	4	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
3	1	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
4	9	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

5 rows × 785 columns

```
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Pixel Values 0', fontsize = 15)
ax.set_ylabel('Pixel Values 1', fontsize = 15)
ax.set_title('2 Pixel Values', fontsize = 20)
targets = [0,1,2,3,4,5,6,7,8,9]
#colors = ['r', 'g', 'b']
from matplotlib import cm


colors = cm.rainbow(np.linspace(0, 1, 10))
```

```
for target, color in zip(targets,colors):
    indicesToKeep = pixel_df['pred_class'] == target
    ax.scatter(pixel_df.loc[indicesToKeep, 'pix_val_77']
               , pixel_df.loc[indicesToKeep, 'pix_val_78']
               , c = color.reshape(1,-1)
               , s = 50)
ax.legend(targets)
ax.grid()
```