CNN/ConvNet

The MNIST database comprises 60,000 training images and 10,000 testing images, sourced from American Census Bureau employees and American high school students. Renowned for image classification tasks, the MNIST dataset is widely accessible through various platforms, including direct access from Tensorflow and Keras APIs. In a previous post, I employed a Neural Network (NN) to demonstrate accuracy using this same dataset. However, for this project, I incorporated Convolutional Neural Networks (CNN) alongside NN to achieve improved outputs.

The MNIST Dataset

The MNIST dataset is a popular dataset for image classification machine learning model tutorials. It is conveniently included in the Keras library and ready to be loaded with build-in functions for analysis. The WIKI page of MNIST provides a detailed description of the dataset: https://en.wikipedia.org/wiki/MNIST_database. It contains 70,000 images of handwritten digits from American Census Bureau employees and American high school students. There are 60,000 training images and 10,000 testing images. Each image has a resolution of 28x28, and the numerical pixel values are in greyscale. Each image is represented by a 28x28 matrix with each element of the matrix an integer between 0 and 255. The label of each image is the intended digit of the handwritten image between 0 and 9. We cover the detailed steps to explore the MNIST dataset in the R and Python notebooks. A sample of the dataset is illustrated in the figure below:[2]

link to the wiki page

Firstly, let's select TensorFlow version 2.x in colab

In [2]:
```
%tensorflow_version 2.x
import tensorflow
tensorflow.__version__
```

```
Colab only includes TensorFlow 2.x; %tensorflow_version has no effect.
```

Out[2]:
```
'2.12.0'
```

In [3]:
```python
# Initialize the random number generator

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt # plotting library
%matplotlib inline
from keras.models import Sequential
import tensorflow
from tensorflow.keras.optimizers import Adam # - Works ,RMSprop
from tensorflow.keras.utils import to_categorical, plot_model
from keras import  backend as K
from keras.layers import Dense
from keras.layers.convolutional import Conv2D, MaxPooling2D
# from keras.models import Sequential
from keras.layers import Activation, Flatten

import random
random.seed(0)

# Ignore the warnings
import warnings
warnings.filterwarnings("ignore")
```

Let's load MNIST dataset

In [4]:
```python
from tensorflow.keras.datasets import mnist

# the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

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

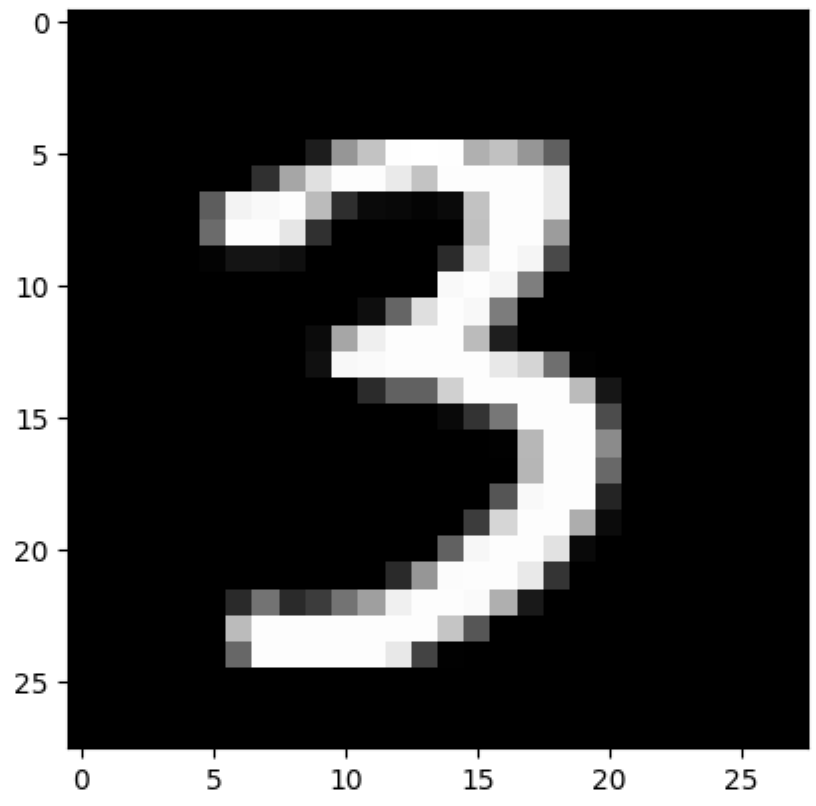
X_train and X_test contain greyscale RGB codes (from 0 to 255) while y_train and y_test contains labels from 0 to 9 which represents which number they actually are.

Let's visualize some numbers using matplotlib

In [5]:
```python
import matplotlib.pyplot as plt
%matplotlib inline
print("Label: {}".format(y_train[10000]))
plt.imshow(X_train[10000], cmap='gray')
```

```
Label: 3
```

Out[5]:
```
<matplotlib.image.AxesImage at 0x7944162b3400>
```



Data Preprocessing

Print shape of the data

In [6]:
```
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

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

Reshape train and test sets into compatible shapes

- Sequential model in tensorflow.keras expects data to be in the format (n_e, n_h, n_w, n_c)
- n_e= number of examples, n_h = height, n_w = width, n_c = number of channels
- do not reshape labels

In [7]:
```
X_train.shape[0]
```

Out[7]:
```
60000
```

In [8]:
```
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)
X_test = X_test.reshape(X_test.shape[0], 28, 28, 1)
```

In [9]:
```
X_train.shape
```

Out[9]:
```
(60000, 28, 28, 1)
```

Normalize data

- we must normalize our data as it is always required in neural network models
- we can achieve this by dividing the RGB codes with 255 (which is the maximum RGB code minus the minimum RGB code)
- normalize X_train and X_test
- make sure that the values are float so that we can get decimal points after division

In [10]:
```
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

X_train /= 255
X_test /= 255
```

In [11]:
```
200/255
```

Out[11]:
```
0.7843137254901961
```

Print shape of data and number of images

- print shape of X_train
- print number of images in X_train
- print number of images in X_test

In [12]:
```
print("X_train shape:", X_train.shape)
print("Images in X_train:", X_train.shape[0])
print("Images in X_test:", X_test.shape[0])
print("Max value in X_train:", X_train.max())
print("Min value in X_train:", X_train.min())
```

```
X_train shape: (60000, 28, 28, 1)
Images in X_train: 60000
Images in X_test: 10000
Max value in X_train: 1.0
Min value in X_train: 0.0
```

One-hot encode the class vector

- convert class vectors (integers) to binary class matrix
- convert y_train and y_test
- number of classes: 10
- we are doing this to use categorical_crossentropy as loss

In [13]:
```
from tensorflow.keras.utils import to_categorical

y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)

# print("Shape of y_train:", y_train.shape)
# print("One value of y_train:", y_train[0])
```

Building Convolutional Neural Network : ConvNet/CNN

Initialize a sequential model again

- define a sequential model
- add 2 convolutional layers
    - no of filters: 32
    - kernel size: 3x3
    - activation: "relu"
    - input shape: (28, 28, 1) for first layer
- flatten the data
    - add Flatten later
    - flatten layers flatten 2D arrays to 1D array before building the fully connected layers
- add 2 dense layers
    - number of neurons in first layer: 128
    - number of neurons in last layer: number of classes
    - activation function in first layer: relu
    - activation function in last layer: softmax
    - we may experiment with any number of neurons for the first Dense layer; however, the final Dense layer must have neurons equal to the number of output classes

In [14]:
```
from tensorflow.keras.layers import Conv2D

model = Sequential()
model.add(Conv2D(filters=32, kernel_size=3, activation="relu", input_shape=(28, 28, 1)))
model.add(Conv2D(filters=32, kernel_size=3, activation="relu"))
model.add(Flatten())
model.add(Dense(128, activation="relu"))
model.add(Dense(10, activation="softmax"))
```

Compile and fit the model

- let's compile our model
    - loss: "categorical_crossentropy"
    - metrics: "accuracy"
    - optimizer: "adam"
- then next step will be to fit model
    - give train data - training features and labels
    - batch size: 32
    - epochs: 10
    - give validation data - testing features and labels

In [29]:
```
# Compile the model
model.compile(loss="categorical_crossentropy", metrics=["accuracy"], optimizer="adam")

# Fit the model
H = model.fit( x=X_train, y=y_train, batch_size=32, epochs=10, validation_split = 0.3)
```

```
Epoch 1/10
1313/1313 [==============================] - 9s 6ms/step - loss: 0.0230 - accuracy: 0.99
20 - val_loss: 0.0189 - val_accuracy: 0.9937
Epoch 2/10
1313/1313 [==============================] - 7s 5ms/step - loss: 0.0153 - accuracy: 0.99
47 - val_loss: 0.0243 - val_accuracy: 0.9928
Epoch 3/10
1313/1313 [==============================] - 7s 6ms/step - loss: 0.0131 - accuracy: 0.99
```

```
57 - val_loss: 0.0239 - val_accuracy: 0.9929
Epoch 4/10
1313/1313 [==============================] - 6s 5ms/step - loss: 0.0109 - accuracy: 0.99
63 - val_loss: 0.0252 - val_accuracy: 0.9926
Epoch 5/10
1313/1313 [==============================] - 7s 6ms/step - loss: 0.0086 - accuracy: 0.99
70 - val_loss: 0.0296 - val_accuracy: 0.9917
Epoch 6/10
1313/1313 [==============================] - 7s 5ms/step - loss: 0.0079 - accuracy: 0.99
73 - val_loss: 0.0350 - val_accuracy: 0.9910
Epoch 7/10
1313/1313 [==============================] - 7s 5ms/step - loss: 0.0076 - accuracy: 0.99
73 - val_loss: 0.0305 - val_accuracy: 0.9922
Epoch 8/10
1313/1313 [==============================] - 8s 6ms/step - loss: 0.0052 - accuracy: 0.99
84 - val_loss: 0.0329 - val_accuracy: 0.9923
Epoch 9/10
1313/1313 [==============================] - 7s 6ms/step - loss: 0.0071 - accuracy: 0.99
75 - val_loss: 0.0363 - val_accuracy: 0.9925
Epoch 10/10
1313/1313 [==============================] - 7s 6ms/step - loss: 0.0057 - accuracy: 0.99
80 - val_loss: 0.0303 - val_accuracy: 0.9927
```

Final loss and accuracy

In [30]:
```
loss, acc = model.evaluate(X_test, y_test)
print("\nTest accuracy: %.1f%%" % (100.0 * acc))
```

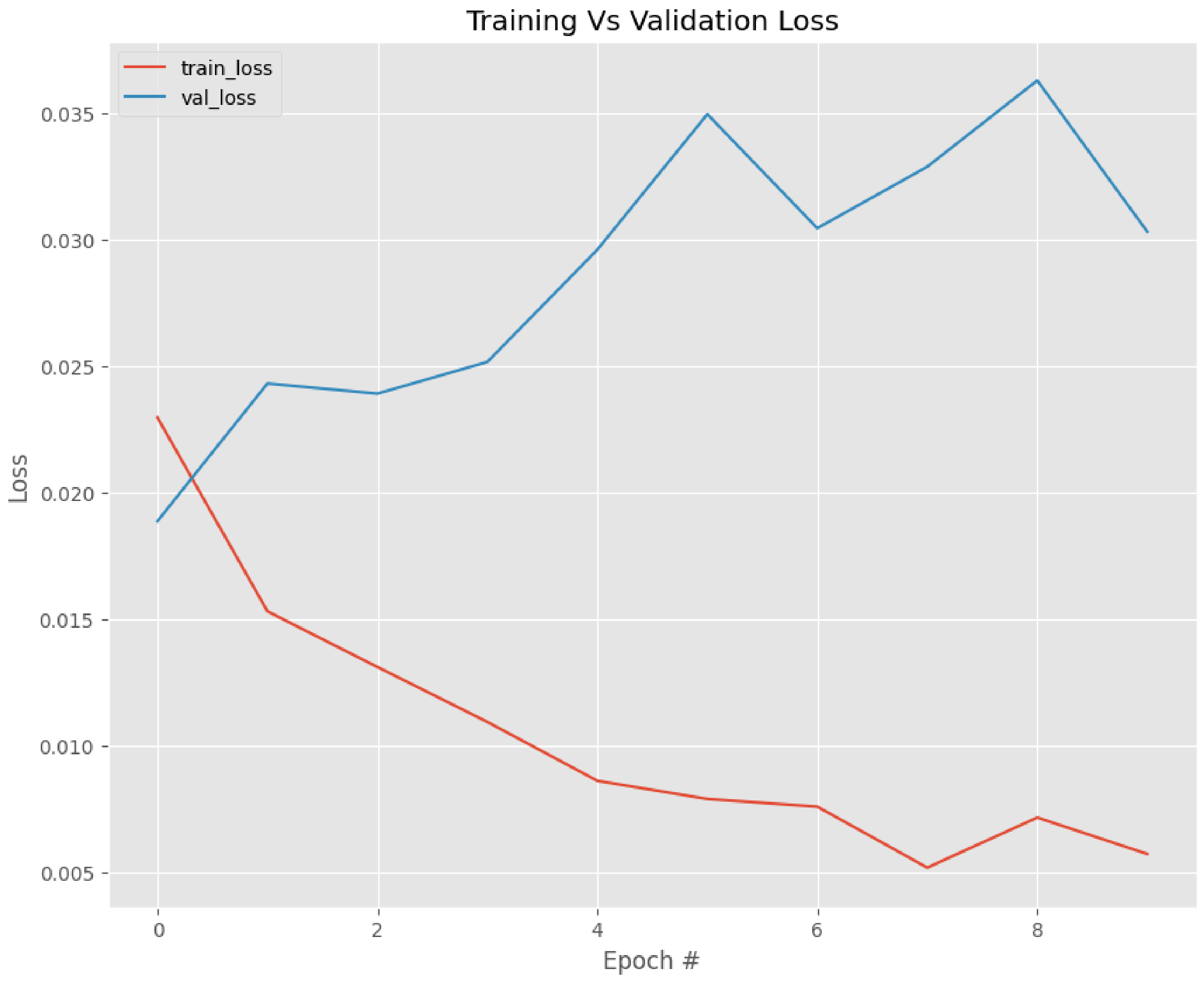
```
313/313 [==============================] - 1s 3ms/step - loss: 0.0417 - accuracy: 0.9912

Test accuracy: 99.1%
```

In [31]:
```
loss, acc = model.evaluate(X_train, y_train)
print("\nTraining accuracy: %.1f%%" % (100.0 * acc))
```

```
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0102 - accuracy: 0.99
75

Training accuracy: 99.8%
```

In [32]:
```
import matplotlib.pyplot as plt


epochs = np.arange(0, len(H.history["loss"]))
plt.figure(figsize=(10,8))
plt.style.use("ggplot")


# plot training and validation loss
plt.style.use("ggplot")
plt.plot(epochs, H.history["loss"], label="train_loss")
plt.plot(epochs, H.history["val_loss"], label="val_loss")
plt.title("Training Vs Validation Loss")
plt.xlabel("Epoch #")
plt.ylabel("Loss")
plt.legend()
```

Out[32]:
```
<matplotlib.legend.Legend at 0x7943a07e4670>
```



Vanilla CNN + Pooling + Dropout + Regularization

Initialize a sequential model again

- define a sequential model
- add 2 convolutional layers
    - no of filters: 32
    - kernel size: 3x3
    - activation: "relu"
    - input shape: (28, 28, 1) for first layer
- add a max pooling layer of size 2x2
- add a dropout layer
    - dropout layers fight with the overfitting by disregarding some of the neurons while training
    - use dropout rate 0.2
- flatten the data
    - add Flatten later
    - flatten layers flatten 2D arrays to 1D array before building the fully connected layers
- add 2 dense layers
    - number of neurons in first layer: 128
    - number of neurons in last layer: number of classes

    - activation function in first layer: relu
    - activation function in last layer: softmax
    - we may experiment with any number of neurons for the first Dense layer; however, the final Dense layer must have neurons equal to the number of output classes

In [46]:
```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dropout, MaxPooling2D

# Initialize the model
model = Sequential()

# Add a Convolutional Layer with 32 filters of size 3X3 and activation function as 'relu
model.add(Conv2D(filters=32, kernel_size=3, activation="relu", input_shape=(28, 28, 1)))

# Add a Convolutional Layer with 32 filters of size 3X3 and activation function as 'relu
model.add(Conv2D(filters=32, kernel_size=3, activation="relu"))

# Add a MaxPooling Layer of size 2X2
model.add(MaxPooling2D(pool_size=(2, 2)))

# Apply Dropout with 0.2 probability
model.add(Dropout(rate=0.2))

# Flatten the layer
model.add(Flatten())

# Add Fully Connected Layer with 128 units and activation function as 'relu'
model.add(Dense(128, activation="relu")) # you can add regularization also : kernel_regu
model.add(Dropout(rate=0.2))
#Add Fully Connected Layer with 10 units and activation function as 'softmax'
model.add(Dense(10, activation="softmax"))
```

Compile and fit the model

- let's compile our model
    - loss: "categorical_crossentropy"
    - metrics: "accuracy"
    - optimizer: "adam"
- Use EarlyStopping
- then next step will be to fit model
    - give train data - training features and labels
    - batch size: 32
    - epochs: 10
    - give validation data - testing features and labels

In [47]:
```python
# Compile the model
model.compile(loss="categorical_crossentropy", metrics=["accuracy"], optimizer="adam")

# Use earlystopping
callback = tensorflow.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2,

# Fit the model
H = model.fit(x=X_train, y=y_train, batch_size=32, epochs=10, validation_data=(X_test, y
```

```
Epoch 1/10
1875/1875 [==============================] - 10s 4ms/step - loss: 0.1480 - accuracy: 0.9
554 - val_loss: 0.0520 - val_accuracy: 0.9828
Epoch 2/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.0553 - accuracy: 0.98
```

```
32 - val_loss: 0.0364 - val_accuracy: 0.9882
Epoch 3/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.0394 - accuracy: 0.98
80 - val_loss: 0.0364 - val_accuracy: 0.9883
```

Final loss and accuracy

In [48]:
```
loss, acc = model.evaluate(X_test, y_test)
print("\nTest accuracy: %.1f%%" % (100.0 * acc))
```

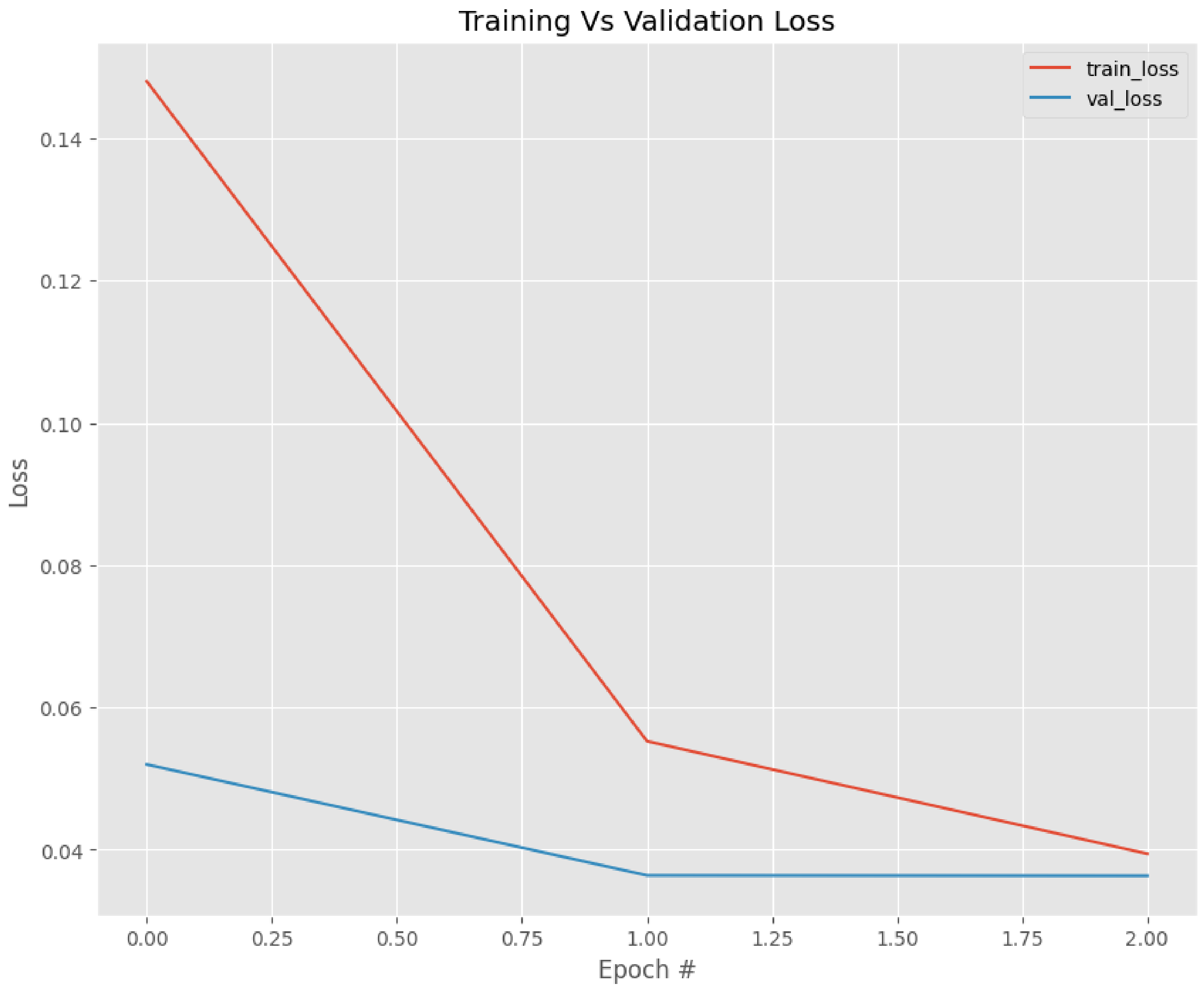
```
313/313 [==============================] - 1s 3ms/step - loss: 0.0364 - accuracy: 0.9883

Test accuracy: 98.8%
```

In [49]:
```
loss, acc = model.evaluate(X_train, y_train)
print("\nTraining accuracy: %.1f%%" % (100.0 * acc))
```

```
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0170 - accuracy: 0.99
46

Training accuracy: 99.5%
```

In [50]:
```
import matplotlib.pyplot as plt


epochs = np.arange(0, len(H.history["loss"]))
plt.figure(figsize=(10,8))
plt.style.use("ggplot")


# plot training and validation loss
plt.style.use("ggplot")
plt.plot(epochs, H.history["loss"], label="train_loss")
plt.plot(epochs, H.history["val_loss"], label="val_loss")
plt.title("Training Vs Validation Loss")
plt.xlabel("Epoch #")
plt.ylabel("Loss")
plt.legend()
```

Out[50]:
```
<matplotlib.legend.Legend at 0x794374958fd0>
```

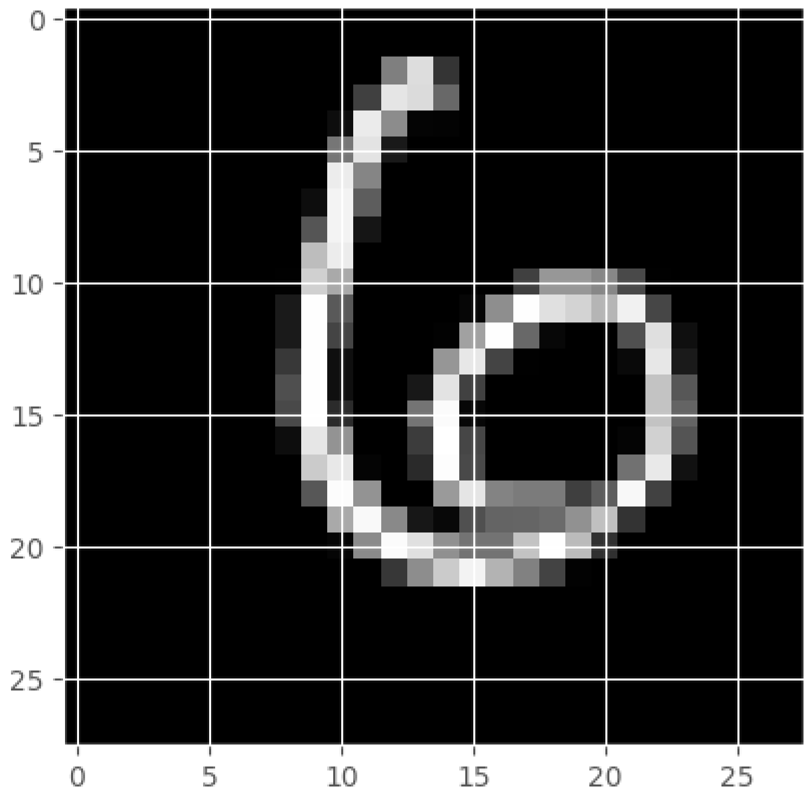


Make Prediction

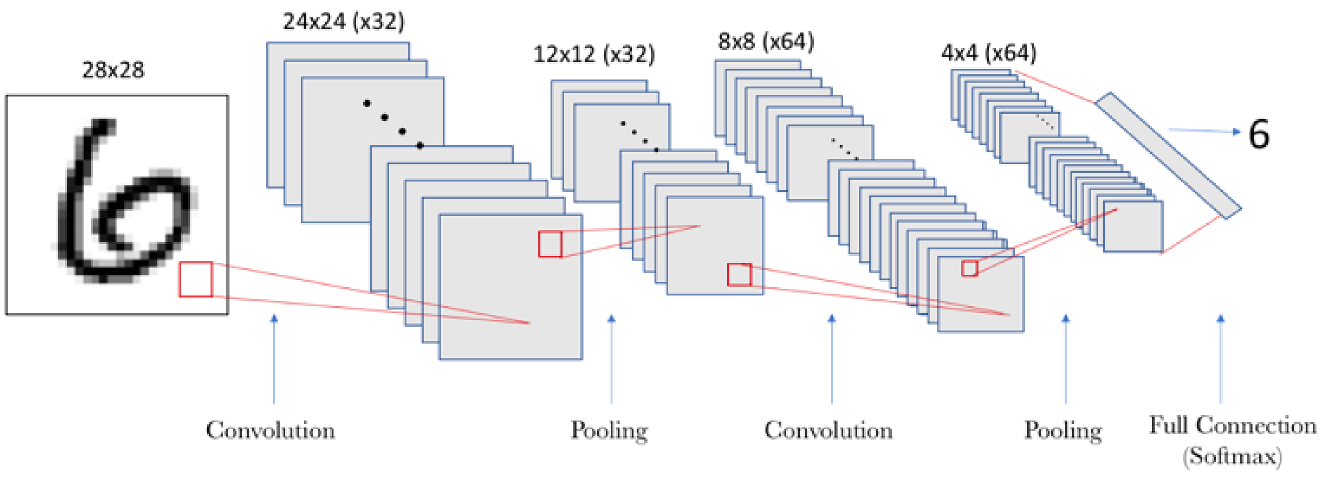
Let's visualize results using matplotlib

In [58]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.imshow(X_test[100].reshape(28, 28), cmap='gray')
y_pred = model.predict(X_test[100].reshape(1, 28, 28, 1)) #No.of images, Height, width,
print("Predicted label:", y_pred.argmax())
print("Softmax Outputs:", y_pred)
print(y_pred.sum())
```

```
1/1 [==============================] - 0s 97ms/step
Predicted label: 6
Softmax Outputs: [[1.07105400e-07 1.46991335e-08 2.16243080e-13 2.40382174e-14
  4.57312382e-07 1.50646207e-09 9.99999404e-01 3.41020116e-12
  4.15835046e-08 2.76238667e-12]]
1.0
```



Architecture of what happened internally is shown below:



```
In [ ]:
```