Project-1: Building Convolutional Neural Networks on MNIST Dataset

By Vivek Prakash Upreti

thevivekai@gmail.com

Learning Objectives

- Understand CNN architecture and its components
- Implement CNN from scratch using TensorFlow/Keras
- Master data preprocessing for image classification
- Compare different CNN architectures

Dataset Information

- 70,000 grayscale images of handwritten digits (0-9)
- Training set: 60,000 images
- Test set: 10,000 images
- Image dimensions: 28x28 pixels
- 10 classes (digits 0-9)

Step-1: Import the Required Libraries

In [57]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D,MaxPooling2D,Flatten,Dense,Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import classification_report,confusion_matrix
import pandas as pd
```

Step-2: Data Loading and Pre Processing

2.1 Data Loading & Exploration

In [58]:
```
# Here we Load the MNIST dataset using tf.keras.datasets.mnist.load_data()
(x_train,y_train),(x_test,y_test)=tf.keras.datasets.mnist.load_data()
```

In [59]:
```
# Here we Display the shape of training and testing sets
x_train.shape,x_test.shape,y_train.shape,y_test.shape
```

Out[59]:
```
((60000, 28, 28), (10000, 28, 28), (60000,), (10000,))
```

In [60]:
```
# One Hot Encode Label
y_train_cat=to_categorical(y_train,10)
y_test_cat=to_categorical(y_test,10)
```

2.2 Data Pre-processing

- First we normalize the data with in a range by which all value come in a certain range
- Then we reshaping the image to add channel dimension(28,28,1)

In [61]:
```
# Normalization
x_train,x_test=x_train/255.0,x_test/255.0
```

In [62]:
```
# Reshaping
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
```

Step-3: Building Basic CNN Architecture

3.1 Design CNN Architecture:

In [63]:
```
model = Sequential()
model.add(Conv2D(32,(3,3),activation='relu',input_shape=[28,28,1],padding='same'))
model.add(MaxPooling2D((2,2)))
model.add(Conv2D(64,(3,3),activation='relu'))
model.add(MaxPooling2D((2,2)))
model.add(Conv2D(64,(3,3),activation='relu'))
model.add(Flatten())
model.add(Dense(64,activation='relu'))
model.add(Dense(10,activation='softmax'))
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/convolutional/base_conv.py:113: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

In [64]:
```
model.summary()
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
conv2d_9 (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d_6 (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_10 (Conv2D)	(None, 12, 12, 64)	18,496
max_pooling2d_7 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_11 (Conv2D)	(None, 4, 4, 64)	36,928
flatten_3 (Flatten)	(None, 1024)	0
dense_6 (Dense)	(None, 64)	65,600
dense_7 (Dense)	(None, 10)	650

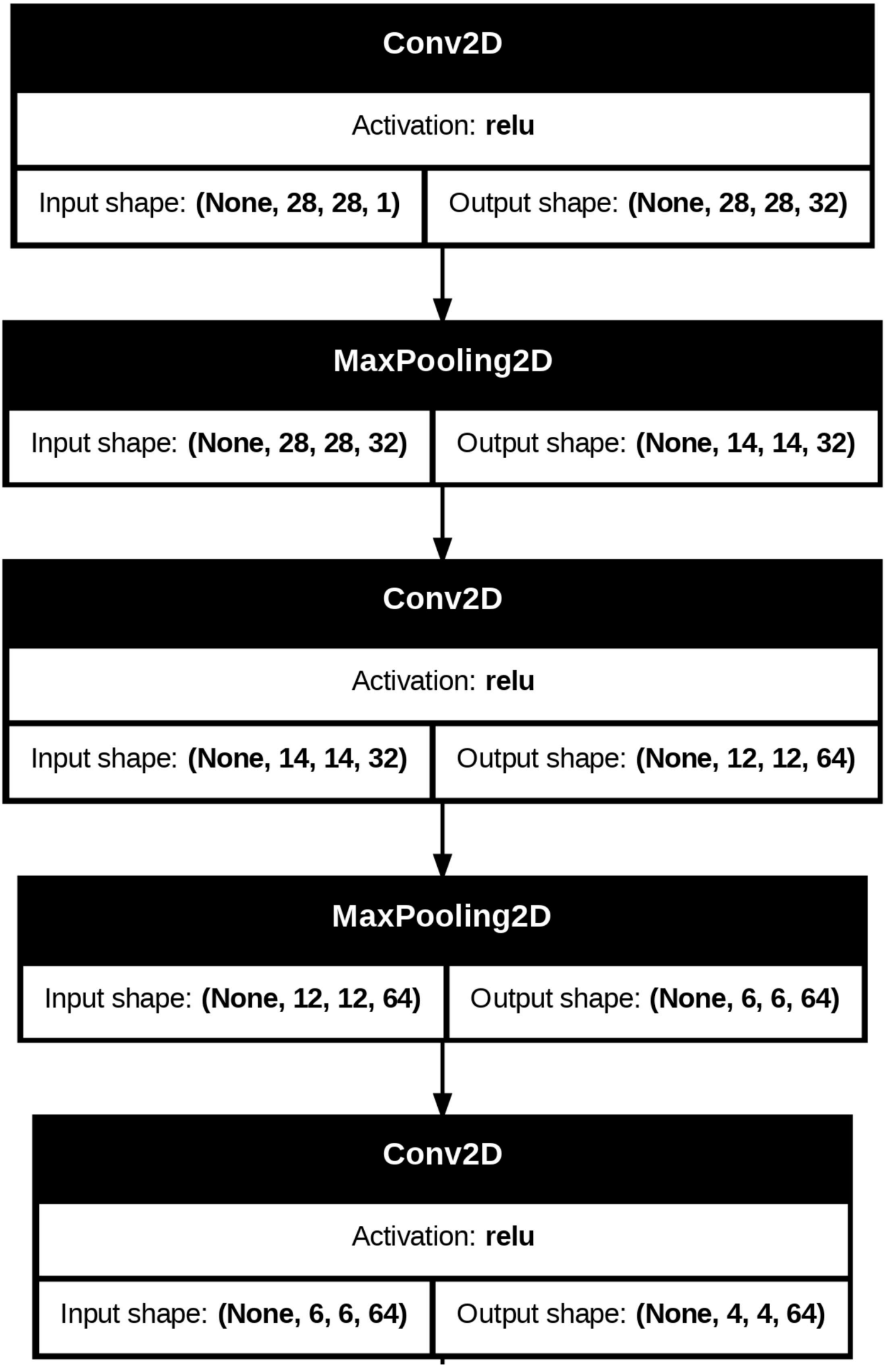
Total params: 121,994 (476.54 KB)

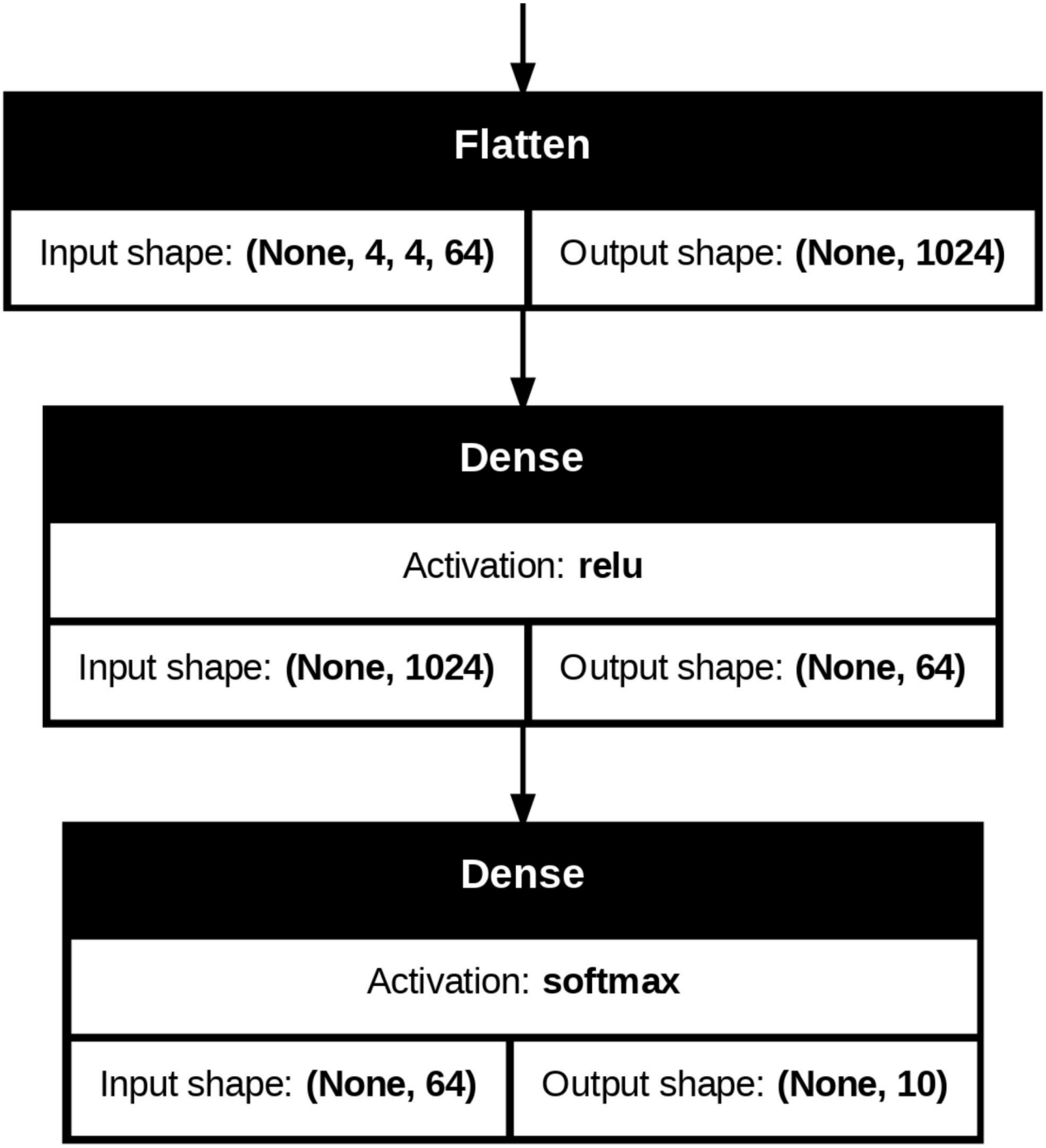
Trainable params: 121,994 (476.54 KB)

Non-trainable params: 0 (0.00 B)

In [65]:
```
tf.keras.utils.plot_model(model,to_file='model_architecture.png',show_shapes=True,show_layer_activations=True)
```

Out[65]:


Conv2D
Activation: relu
Input shape: (None, 28, 28, 1)
Output shape: (None, 28, 28, 32)
MaxPooling2D
Input shape: (None, 28, 28, 32)
Output shape: (None, 14, 14, 32)
Conv2D
Activation: relu
Input shape: (None, 14, 14, 32)
Output shape: (None, 12, 12, 64)
MaxPooling2D
Input shape: (None, 12, 12, 64)
Output shape: (None, 6, 6, 64)
Conv2D
Activation: relu
Input shape: (None, 6, 6, 64)
Output shape: (None, 4, 4, 64)



3.2 Model Compilation

In [66]:
```
model.compile(optimizer=Adam(learning_rate=0.001),loss='categorical_crossentropy',metrics=['accuracy'])
```

Step-4: Model Training & Evaluation

4.1 Model Training

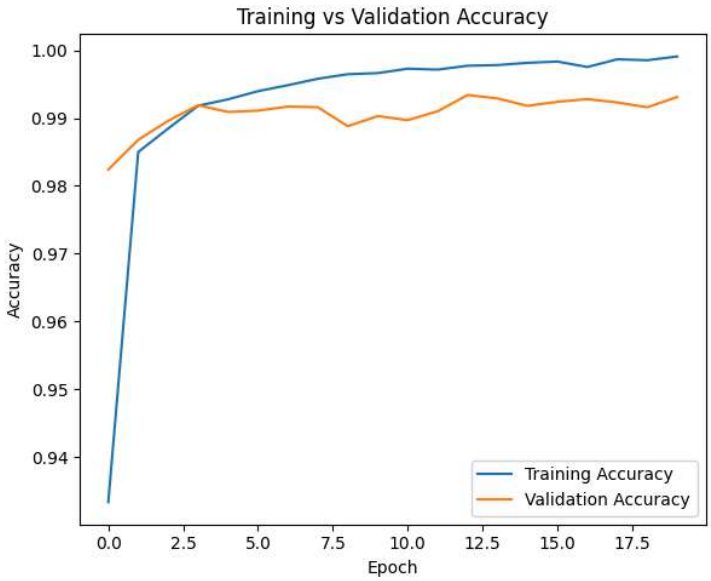
In [67]:
```
mnist_model= model.fit(x_train,y_train_cat,batch_size=128,epochs=20,validation_data=(x_test, y_test_cat))
```

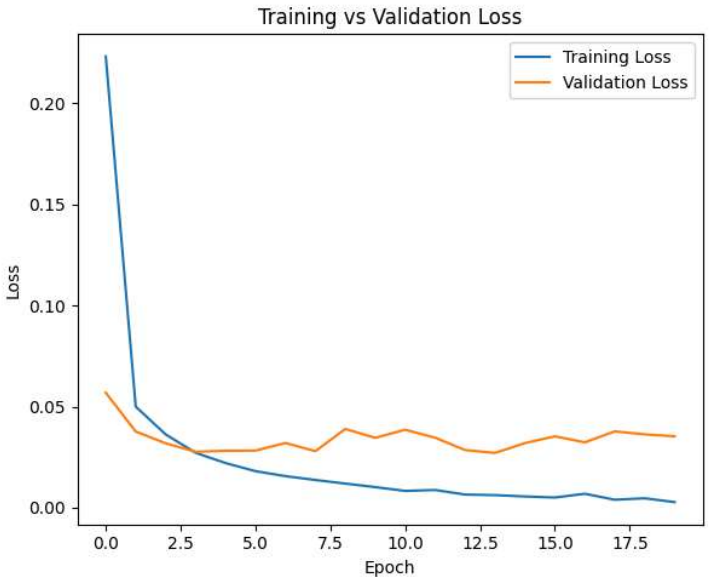
```
Epoch 1/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - accuracy: 0.8394 - loss: 0.5409 - val_accuracy: 0.9824 - val_loss: 0.0569
Epoch 2/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9853 - loss: 0.0506 - val_accuracy: 0.9868 - val_loss: 0.0377
Epoch 3/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9881 - loss: 0.0365 - val_accuracy: 0.9896 - val_loss: 0.0319
Epoch 4/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9921 - loss: 0.0267 - val_accuracy: 0.9919 - val_loss: 0.0277
Epoch 5/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9930 - loss: 0.0210 - val_accuracy: 0.9909 - val_loss: 0.0282
Epoch 6/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9950 - loss: 0.0162 - val_accuracy: 0.9911 - val_loss: 0.0282
Epoch 7/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9953 - loss: 0.0145 - val_accuracy: 0.9917 - val_loss: 0.0320
Epoch 8/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9963 - loss: 0.0128 - val_accuracy: 0.9916 - val_loss: 0.0280
Epoch 9/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9970 - loss: 0.0102 - val_accuracy: 0.9888 - val_loss: 0.0389
Epoch 10/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9971 - loss: 0.0086 - val_accuracy: 0.9903 - val_loss: 0.0346
Epoch 11/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9971 - loss: 0.0087 - val_accuracy: 0.9897 - val_loss: 0.0386
Epoch 12/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.9972 - loss: 0.0087 - val_accuracy: 0.9910 - val_loss: 0.0346
Epoch 13/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.9979 - loss: 0.0058 - val_accuracy: 0.9934 - val_loss: 0.0285
Epoch 14/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9981 - loss: 0.0056 - val_accuracy: 0.9929 - val_loss: 0.0271
Epoch 15/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9981 - loss: 0.0054 - val_accuracy: 0.9918 - val_loss: 0.0320
Epoch 16/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9984 - loss: 0.0051 - val_accuracy: 0.9924 - val_loss: 0.0353
Epoch 17/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9981 - loss: 0.0056 - val_accuracy: 0.9928 - val_loss: 0.0323
Epoch 18/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9989 - loss: 0.0033 - val_accuracy: 0.9923 - val_loss: 0.0377
Epoch 19/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9988 - loss: 0.0039 - val_accuracy: 0.9916 - val_loss: 0.0363
Epoch 20/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9989 - loss: 0.0029 - val_accuracy: 0.9931 - val_loss: 0.0353
```

Since Our Training Accuracy & Testing both are approximately same so it means our model is perfectly trained

In [68]:

```python
# Plot training history:

plt.figure(figsize=(12, 5))

# Plotting the accuracy
plt.subplot(1, 2, 1)
plt.plot(mnist_model.history['accuracy'], label='Training Accuracy')
plt.plot(mnist_model.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training vs Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

# Plotting the loss
plt.subplot(1, 2, 2)
plt.plot(mnist_model.history['loss'], label='Training Loss')
plt.plot(mnist_model.history['val_loss'], label='Validation Loss')
plt.title('Training vs Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()
```





4.2 Model Evaluation

In [69]:
```
# Evaluate model on test set
test_loss,test_acc=model.evaluate(x_test,y_test_cat)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9909 - loss: 0.0457
```

In [70]:
```
print('Overall Test Accuracy',test_acc)
print('Overall Test Loss',test_loss)
```

```
Overall Test Accuracy 0.9930999875068665
Overall Test Loss 0.035328932106494904
```

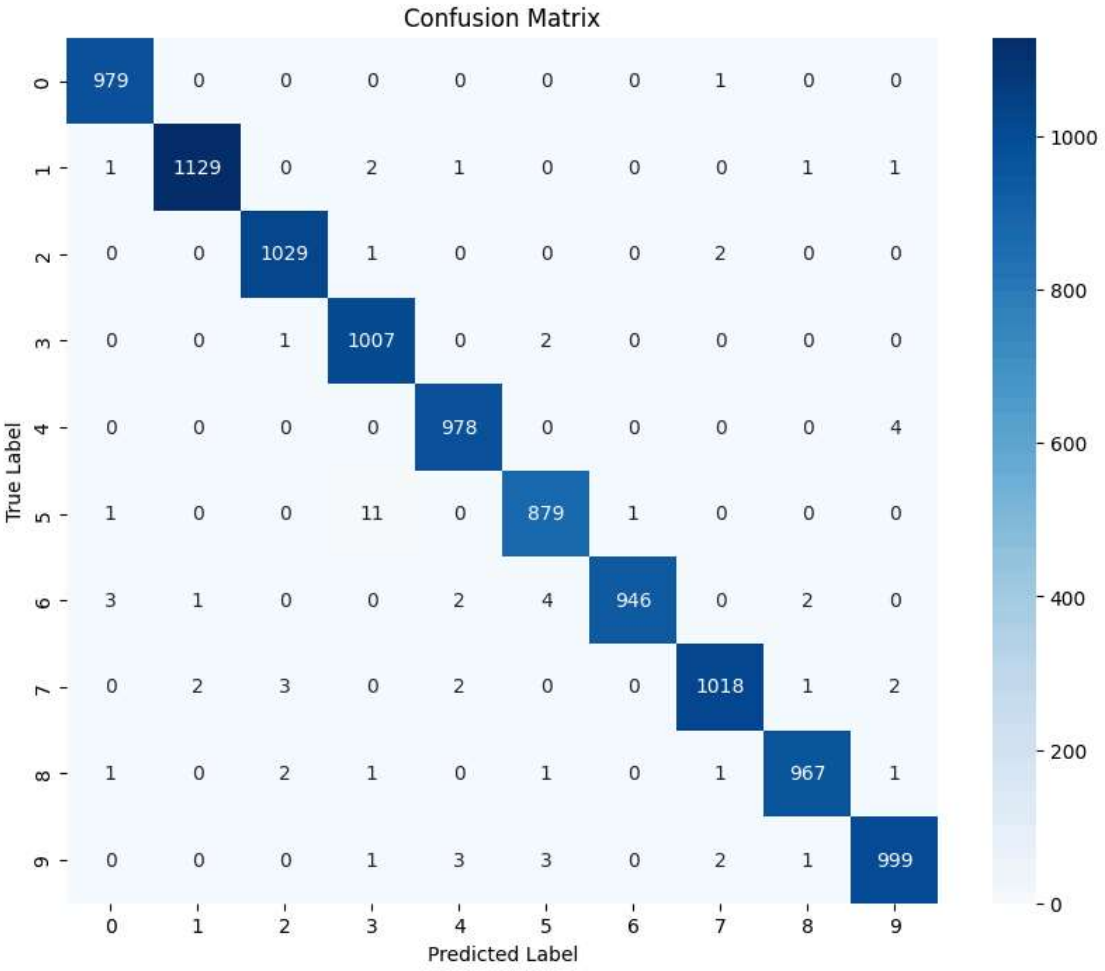
In [71]:
```
# Generate classification report
# Get the model's predictions (probability arrays)
predictions = model.predict(x_test)

# Convert probability arrays to single integer labels
predicted_labels = np.argmax(predictions, axis=1)

# Convert one-hot encoded true labels to single integer labels
true_labels = np.argmax(y_test_cat, axis=1)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
```

In [72]:
```
print("Classification Report:")
print(classification_report(true_labels, predicted_labels, digits=4))
```

```
Classification Report:
              precision    recall  f1-score   support

           0     0.9939    0.9990    0.9964       980
           1     0.9973    0.9947    0.9960      1135
           2     0.9942    0.9971    0.9956      1032
           3     0.9844    0.9970    0.9907      1010
           4     0.9919    0.9959    0.9939       982
           5     0.9888    0.9854    0.9871       892
           6     0.9989    0.9875    0.9932       958
           7     0.9941    0.9903    0.9922      1028
           8     0.9949    0.9928    0.9938       974
           9     0.9921    0.9901    0.9911      1009

    accuracy                         0.9931     10000
   macro avg     0.9930    0.9930    0.9930     10000
weighted avg     0.9931    0.9931    0.9931     10000
```

In [73]:
```
# Create and Visualize Confusion Matrix
# Generate the confusion matrix
cm = confusion_matrix(true_labels, predicted_labels)

# Visualize the confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```



Step-5: Conclusion

Based on the analysis and the results obtained, here are the key takeaways from this project:

- **High Accuracy**: The CNN model achieved an impressive overall test accuracy of approximately 0.98. This indicates that the model is highly effective in correctly classifying handwritten digits.
- **Low Loss**: The overall test loss was approximately 0.04, suggesting that the model's predictions are very close to the true labels on the test dataset.
- **Effective Training**: The training and validation accuracy curves show a consistent increase, with both reaching similar high values by the end of training. This indicates that the model is not overfitting and generalizes well to unseen data. Similarly, the training and validation loss curves decrease steadily, suggesting that the model learned effectively.
- **Robust Performance across Classes**: The classification report shows high precision, recall, and f1-scores for each digit class (0-9). This demonstrates the model's robust performance across all categories and its ability to accurately identify individual digits.
- **Clear Separation of Classes**: The confusion matrix visually confirms the model's strong performance. The diagonal elements, representing correct classifications, are significantly larger than the off-diagonal elements, which represent misclassifications. This visually reinforces the high accuracy and low error rate.

In summary, the developed CNN model demonstrates excellent performance on the MNIST dataset, effectively classifying handwritten digits with high accuracy and low loss. The visualizations of the training history and the confusion matrix further validate the model's effectiveness and generalization capabilities.

In [73]: