Experiment No 9

Title: MNIST digit classification before and after shuffling Train CNN on Original Data Train CNN on shuffled data

Aim: To implement MNIST digit classification before and after shuffling Train CNN on Original Data Train CNN on shuffled data.

Theory:

MNIST Dataset:

The MNIST dataset consists of 28x28 grayscale images of handwritten digits (0-9). Each image is a grid of pixel values representing the intensity of ink at each pixel location.

Convolutional Neural Networks (CNNs):

- **CNNs for Image Classification**: CNNs are effective for image classification due to their ability to capture spatial hierarchies and extract features from images.
- **Training a CNN**: CNNs consist of convolutional layers (for feature extraction), pooling layers (for down sampling), and fully connected layers (for classification). The network is trained to minimize a loss function (e.g., cross-entropy) using backpropagation and optimizers like Adam.
- **Original Data Training**: When trained on the original MNIST data, the CNN learns patterns from images presented in their natural order.
- **Shuffled Data Training**: Shuffling the MNIST data randomizes the order of image presentation during training, potentially making the learning task more challenging by disrupting any inherent data order.
- **Effects of Shuffling**: Data shuffling introduces variability, helping the model generalize better to unseen data and preventing overfitting to the dataset's inherent patterns.
- **Performance Comparison**: Training on shuffled data may improve generalization, but it could require more training epochs to reach the same accuracy compared to the model trained on original, unshuffled data.

Conclusion:

Training a CNN on both original and shuffled data allows us to observe the impact of data order on the learning process.

Shuffling data is a common practice to ensure robustness and generalization in machine learning models, particularly for datasets with inherent order or bias.

The choice between shuffled and original data depends on the problem's requirements, and the experiment highlights the importance of data preprocessing in deep learning.

CNN on Mnist Dataset

Import necessary libraries

In [1]:
```python
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
```

Load and preprocess the MNIST dataset

In [2]:
```python
# Load the MNIST dataset
(X_train, Y_train), (X_test, Y_test) = datasets.mnist.load_data()
```

In [3]:
```python
# Normalize pixel values to be between 0 and 1
X_train, X_test = X_train / 255.0, X_test / 255.0
```

In [4]:
```python
# Reshape the dataset to include the channel dimension
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))
```

Build the CNN model

In [14]:
```python
# Build a CNN model
model = models.Sequential([
    # First convolutional layer with 32 filters, 3x3 kernel, and ReLU activation
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),  # Max pooling layer
    # Second convolutional layer
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),  # Max pooling layer
    # Third convolutional layer
    layers.Conv2D(64, (3, 3), activation='relu'),

    # Flatten the feature maps and connect to a dense layer
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')  # Output layer for 10 digits
])
```

In [13]:
```python
# Print the model summary
model.summary()
```

```
Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
┃ Layer (type)                       ┃ Output Shape          ┃        Pa
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
│ conv2d (Conv2D)                    │ (None, 26, 26, 32)    │
├────────────────────────────────────┼───────────────────────┼──────────
│ max_pooling2d (MaxPooling2D)       │ (None, 13, 13, 32)    │
├────────────────────────────────────┼───────────────────────┼──────────
│ conv2d_1 (Conv2D)                  │ (None, 11, 11, 64)    │         1
├────────────────────────────────────┼───────────────────────┼──────────
│ max_pooling2d_1 (MaxPooling2D)     │ (None, 5, 5, 64)      │
├────────────────────────────────────┼───────────────────────┼──────────
│ conv2d_2 (Conv2D)                  │ (None, 3, 3, 64)      │         3
├────────────────────────────────────┼───────────────────────┼──────────
│ flatten (Flatten)                  │ (None, 576)           │
├────────────────────────────────────┼───────────────────────┼──────────
│ dense (Dense)                      │ (None, 64)            │         3
├────────────────────────────────────┼───────────────────────┼──────────
│ dense_1 (Dense)                    │ (None, 10)            │
└────────────────────────────────────┴───────────────────────┴──────────

 Total params: 279,968 (1.07 MB)

 Trainable params: 93,322 (364.54 KB)

 Non-trainable params: 0 (0.00 B)

 Optimizer params: 186,646 (729.09 KB)
```

Compile the model

In [15]:
```python
# Compile the model with an optimizer, loss function, and performance metrics
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Train the model

```
In [7]:  # Train the model on the training data
         history = model.fit(X_train, Y_train, epochs=5,
                             validation_data=(X_test, Y_test))
```

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 24s 12ms/step - accuracy: 0.8933 - loss: 0.3430 -
val_accuracy: 0.9816 - val_loss: 0.0556
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 20s 11ms/step - accuracy: 0.9847 - loss: 0.0493 -
val_accuracy: 0.9854 - val_loss: 0.0439
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 21s 11ms/step - accuracy: 0.9889 - loss: 0.0341 -
val_accuracy: 0.9908 - val_loss: 0.0301
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 19s 10ms/step - accuracy: 0.9918 - loss: 0.0246 -
val_accuracy: 0.9908 - val_loss: 0.0260
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 22s 11ms/step - accuracy: 0.9943 - loss: 0.0178 -
val_accuracy: 0.9913 - val_loss: 0.0278
```

Evaluate the model

```
In [8]:  # Evaluate the model on the test dataset
         test_loss, test_acc = model.evaluate(X_test, Y_test, verbose=2)
         print(f'Test accuracy: {test_acc*100}')
```

```
313/313 - 3s - 8ms/step - accuracy: 0.9913 - loss: 0.0278
Test accuracy: 99.12999868392944
```

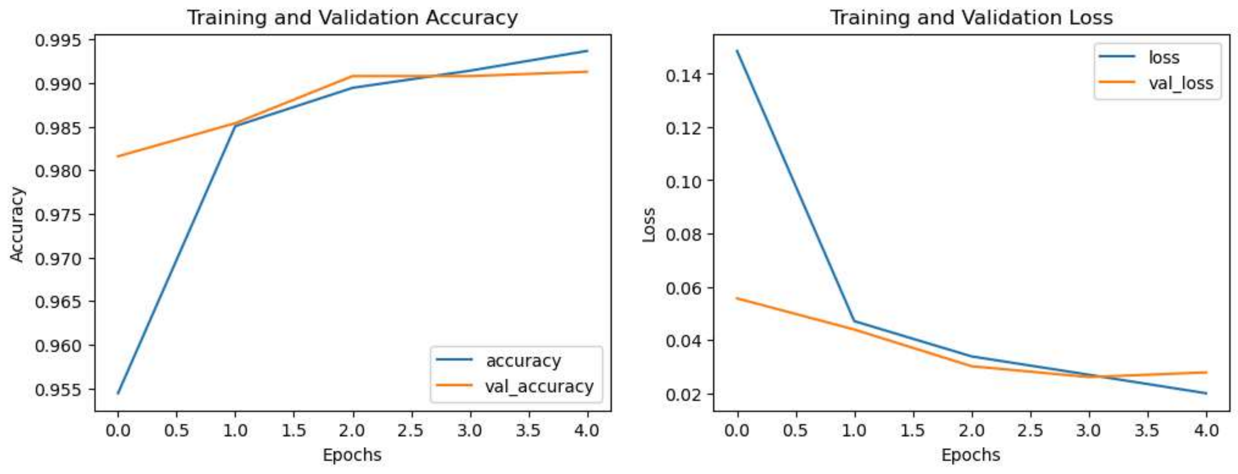
Visualize training results

In [9]:

```
# Plot the accuracy and loss curves
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label='val_accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='loss')
plt.plot(history.history['val_loss'], label='val_loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')

plt.show()
```



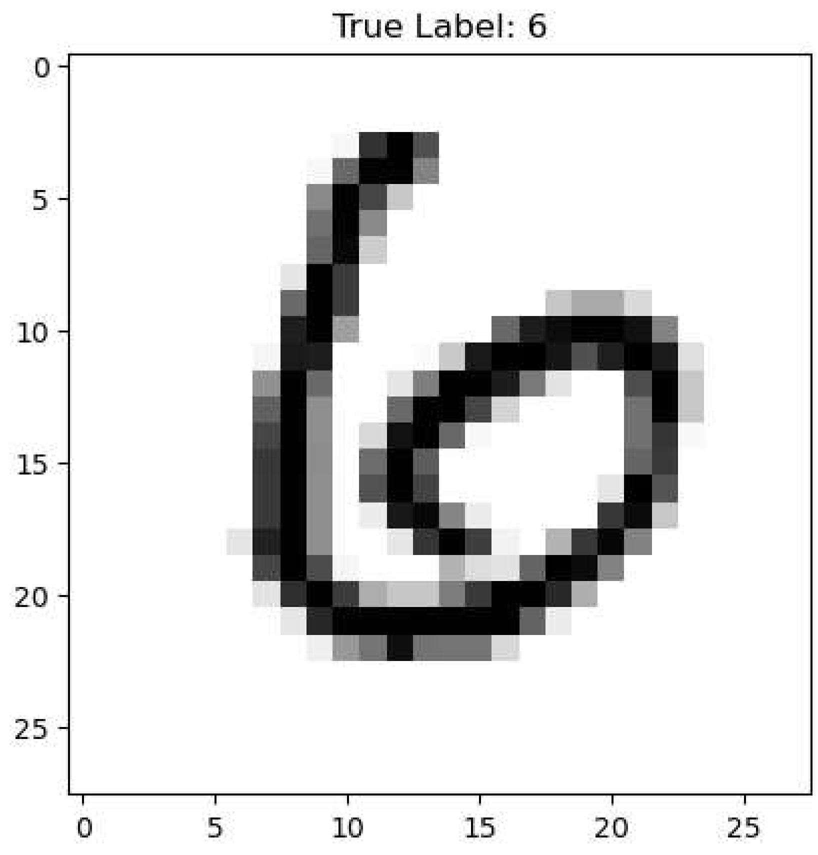
Sample Classification

In [10]:
```python
import numpy as np

# Select a specific example from the test dataset (e.g., the 10th image)
example_index = 11  # You can change this index to any number between 0 and 9999

# Extract the selected test image and its true label
example_image = X_test[example_index]
true_label = Y_test[example_index]

# Display the selected image
plt.imshow(example_image.squeeze(), cmap=plt.cm.binary)
plt.title(f'True Label: {true_label}')
plt.show()
```



In [11]:
```python
# Reshape the image to match the input shape expected by the model
example_image_reshaped = np.expand_dims(example_image, axis=0)  # Add a batch dime

# Use the trained model to predict the class
predicted_probs = model.predict(example_image_reshaped)

# Get the predicted class by finding the index with the highest probability
predicted_label = np.argmax(predicted_probs)

print(f'Predicted Label: {predicted_label}')
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 239ms/step
Predicted Label: 6
```

In [12]:
```python
# Compare the true label with the predicted label
if predicted_label == true_label:
    print(f"Correct! The model predicted {predicted_label} and the true label is 
else:
    print(f"Incorrect! The model predicted {predicted_label} but the true label is
```

```
Correct! The model predicted 6 and the true label is 6.
```