Deep Learning Report

Comparison of Transfer Learning and CNN Model Efficiency
in Recognizing Handwritten Digits

2024

Contents

1 Introduction

1.1 Project Goal

To compare the effectiveness of a classic CNN model and Transfer Learning (ResNet50) in the task of classifying handwritten digits from the MNIST dataset, focusing on metrics like accuracy, training time, and generalization ability.

1.2 Project Significance

Image classification is a key application of Deep Learning, especially in tasks like recognizing patterns such as digits, letters, and objects.

Techniques like CNN and Transfer Learning enable efficient processing of large image datasets, with Transfer Learning offering advantages in scenarios with limited labeled data by reducing the need for extensive training.

1.3 Brief Overview of Methods

- **Convolutional Neural Network (CNN) -** A CNN is composed of convolutional, pooling, and dense layers, designed for image classification tasks. It learns patterns in images, such as edges, making it effective for simple visual tasks like digit classification.
- **Transfer Learning (ResNet50) -** ResNet50 is a deeper architecture pre-trained on ImageNet, using residual (skip) connections to avoid the vanishing gradient problem. It is more complex and capable of learning detailed features, potentially making it better suited for tasks that require greater generalization.
- **Hyperparameter Tuning -** Experiments to optimize the performance of CNN and ResNet50 by varying hyperparameters such as the number of epochs and filter sizes.
- **Long Short-Term Memor (LSTM) -** type of recurrent neural network that models sequential data by capturing temporal dependencies. For this project, each image is treated as a sequence of rows, allowing the model to learn sequential patterns within the data.
- **Transformer Model -** leverages a simplified self-attention mechanism to capture global dependencies in images. Unlike traditional transformers, this implementation treats images as sequences of pixel features without embedding layers, providing an efficient approach for classification tasks.

1.4 Dataset

MNIST – a dataset of images of handwritten digits from 0 to 9, with 70,000 grayscale images sized 28x28 pixels. It is widely used for image classification tasks, split into 60,000 training and 10,000 testing samples.

2 Dataset Description

- **Dataset:** MNIST (Modified National Institute of Standards and Technology) [Kaggle(2021)]

  A widely used dataset for training and testing image processing systems, containing 70,000 images of handwritten digits (0–9).

- **Structure of the Dataset**
  Training Set: 60,000 images for model training.
  Test Set: 10,000 images for evaluating model performance.

- **Image Details:**
  Each image is 28x28 pixels and grayscale (1 color channel).
  Pixel values range from 0 (black) to 255 (white), normalized to [0, 1] for improved processing by neural networks.

3 Data Preparation

```
# Importing libraries
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt
```

Loading the data:
Loads mnist_train.csv and mnist_test.csv datasets.

```
train_data = pd.read_csv('mnist_train.csv')
test_data = pd.read_csv('mnist_test.csv')
```

Separating labels:
The label column is separated from the pixel data.

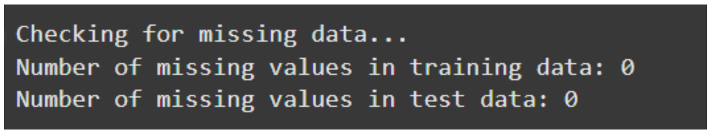
```
train_labels = train_data['label']
train_images = train_data.drop(columns=['label'])

test_labels = test_data['label']
test_images = test_data.drop(columns=['label'])
```

Scaling pixel values:
Converts image data to a numpy array and normalizes pixel values to the [0, 1] range.

```
train_images = train_images.values.reshape(-1, 28, 28, 1) / 255.0
test_images = test_images.values.reshape(-1, 28, 28, 1) / 255.0

train_labels = train_labels.values
test_labels = test_labels.values
```

Checking for missing data

```
print("Checking for missing data...")
missing_train = train_images.size - np.count_nonzero(~np.isnan(train_images))
missing_test = test_images.size - np.count_nonzero(~np.isnan(test_images))

print(f"Number of missing values in training data: {missing_train}")
print(f"Number of missing values in test data: {missing_test}")
```



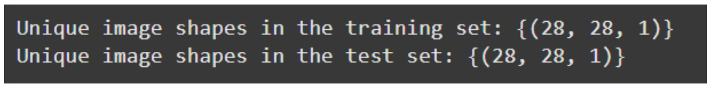
```
Checking for missing data...
Number of missing values in training data: 0
Number of missing values in test data: 0
```

Figure 1: Missing data
[Google(2023)]

Checking dimensional consistency

If the result is just (28, 28, 1) the data is valid.

```
train_shapes = [img.shape for img in train_images]
test_shapes = [img.shape for img in test_images]

unique_train_shapes = set(train_shapes)
unique_test_shapes = set(test_shapes)

print(f"Unique image shapes in the training set: {unique_train_shapes}")
print(f"Unique image shapes in the test set: {unique_test_shapes}")
```



```
Unique image shapes in the training set: {(28, 28, 1)}
Unique image shapes in the test set: {(28, 28, 1)}
```

Figure 2: Checking dimensional consistency
[Google(2023)]

Displaying sample images:

Shows a few sample images with labels to visualize the data.

```
plt.figure(figsize=(10, 10))
for i in range(6):
    plt.subplot(2, 3, i+1)
    plt.imshow(train_images[i].reshape(28, 28), cmap='gray')
    plt.title(f"Label: {train_labels[i]}")
    plt.axis('off')
plt.show()
```

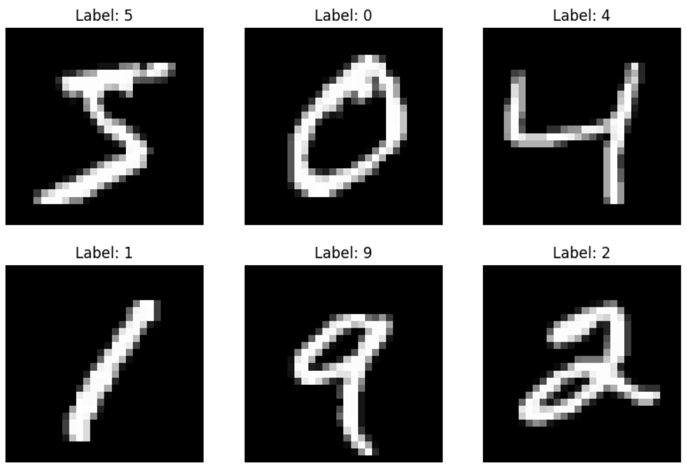


Figure 3: Handwritten numbers - sample images
[Google(2023)]

Preparing data for Transfer Learning (ResNet50):
Resizes images to 32x32 pixels and converts grayscale images to RGB format (3 channels) required by ResNet50.

```
train_images_resnet = np.array([tf.image.grayscale_to_rgb(tf.image.resize(image, (32, 32))) for image in train_images])
test_images_resnet = np.array([tf.image.grayscale_to_rgb(tf.image.resize(image, (32, 32))) for image in test_images])

# Checking the shape of prepared data
print("Image shape for CNN model:", train_images.shape)  # (60000, 28, 28, 1)
print("Image shape for ResNet50 model:", train_images_resnet.shape)  # (60000, 32, 32, 3)
```

4 The Convolutional Neural Network (CNN)

A Convolutional Neural Network (CNN) was designed specifically for classifying handwritten digits in the MNIST dataset.

4.1 Implementation Steps

- **Model Architecture:**
  - **Convolutional Layers -** Extract features from the images by applying filters. The number of filters (32, 64) was chosen to balance computational efficiency and the ability to detect increasingly complex features at different stages of the network.
  - **MaxPooling Layers -** Reduces the spatial dimensions of the feature maps, retaining the most significant information.
  - **Flatten Layer -** Converts 2D feature maps to a 1D vector for input to dense layers.
  - **Dense Layers -** Performs classification with softmax activation in the output layer to assign probabilities to each class.
  - **Kernel Size -** The 3x3 kernel size was chosen as it offers a good trade-off between computational complexity and effective spatial feature detection.

CNN Model Code:

```
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

- **Model Compilation:**
  - **Optimizer:** Adam
  - **Loss Function:** Sparse Categorical Crossentropy
  - **Metrics:** Accuracy

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Adam provides a good balance between speed and stability during training. Sparse Categorical Crossentropy is suitable because the labels are integers, so there is no need to convert them into one-hot encoding.

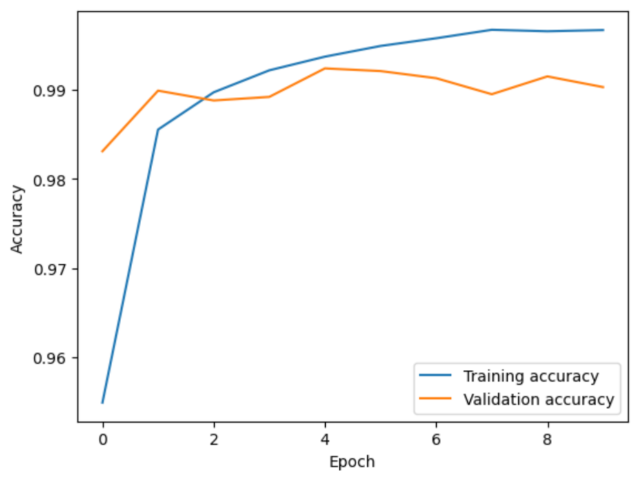
- **Model Summary:**
  The model comprises three convolutional layers followed by pooling layers, and two dense layers for final classification. The total number of trainable parameters in the model is optimized to balance performance and computational efficiency.

- **Training Details**
  - **Dataset ->** MNIST, containing 60,000 training images and 10,000 test images, normalized to [0, 1].
  - **Batch Size ->** 64
  - **Epochs ->** 10
  - **Learning Rate ->** 0.001 (default for Adam). How big steps the model takes when updating weights based on the gradient
  - **Callback ->** Early Stopping: Stops training when validation accuracy does not improve for 3 consecutive epochs.
    Learning Rate Scheduler: Reduces the learning rate by 50% when the validation accuracy plateaus to refine the model's performance.

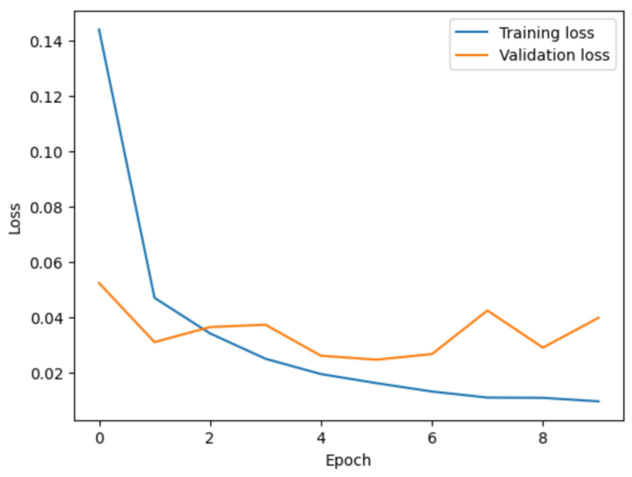
4.2 Results and Observations

The model achieves high accuracy on the MNIST dataset, indicating effective learning from the data. Shows the accuracy of the model on the training and validation datasets, demonstrating the model's learning progress.

4.2.1 Accurancy and Loss graphs



(a) Training and Validation Accuracy [Google(2023)]



(b) Training and Validation Loss [Google(2023)]

- **Accurancy** -> The model achieved a test accuracy of 99.03 %
- **Loss** -> Final loss is 0.051. Training and validation loss steadily decreased, demonstrating the model's ability to generalize well.

4.2.2 Confusion Matrix

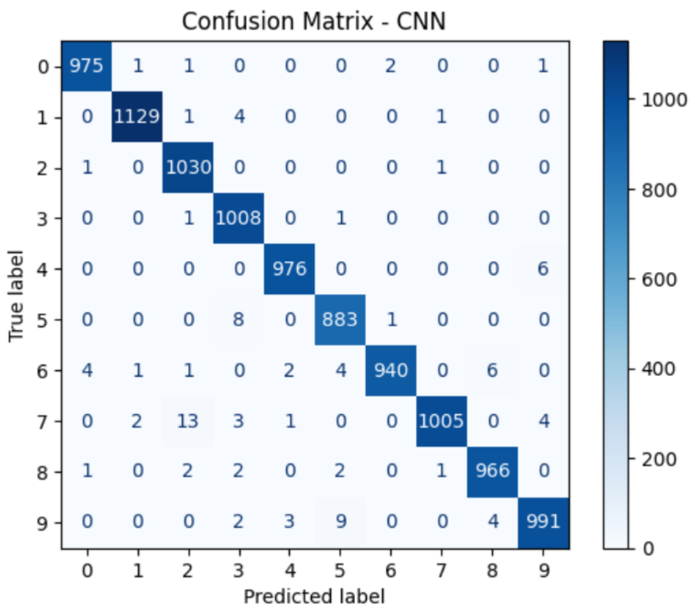


Figure 5: Confusion Matrix for CNN

4.3 Performance Analysis

The graphs illustrate the training and validation accuracy and loss over the epochs, where:

- **Training Accuracy** -> The model quickly converged, reaching near-optimal accuracy within a few epochs.

- **Validation Accuracy** -> Validation accuracy remained stable and close to training accuracy, indicating minimal overfitting.
- **Loss Curves** -> Both training and validation loss decreased steadily, reinforcing the model's stability.

4.4 Conclusion of CNN Model

CNNs are well-suited for image classification tasks due to their ability to capture spatial patterns in images. For the MNIST dataset, the CNN architecture provided a good balance between computational efficiency and classification performance, achieving superior results compared to more complex models.

5 Transfer Learning Model Implementation (ResNet50)

This section presents the implementation of a Transfer Learning model based on ResNet50, pre-trained on the ImageNet dataset. This model was utilized to leverage its feature extraction capabilities for classifying images in our dataset.

5.1 Implementation Steps

- **Loading the Base Model (ResNet50)**
  - We used ResNet50 pre-trained on the ImageNet dataset, excluding the top classification layers to customize it for the MNIST dataset.
  - The base model's weights were set to non-trainable to focus on our custom layers for the classification task.

```
base_model = tf.keras.applications.ResNet50(input_shape=(32, 32, 3), include_top=False, weights='imagenet')
base_model.trainable = False  # freeze base model weights
```

- **Data Preparation for RGB Input**
  - As ResNet50 requires RGB images, grayscale MNIST images were converted to RGB format and resized to 32x32 pixels to match the expected input size.

```
train_images_resnet = np.array([tf.image.grayscale_to_rgb(tf.image.resize(image, (32, 32))) for image in train_images])
test_images_resnet = np.array([tf.image.grayscale_to_rgb(tf.image.resize(image, (32, 32))) for image in test_images])
```

- **Building the Model**
  - Added a Lambda layer to ensure input images are converted to RGB.
  - Used a Global Average Pooling layer to reduce dimensionality of feature maps.
  - Added a Dense layer with softmax activation for final classification.

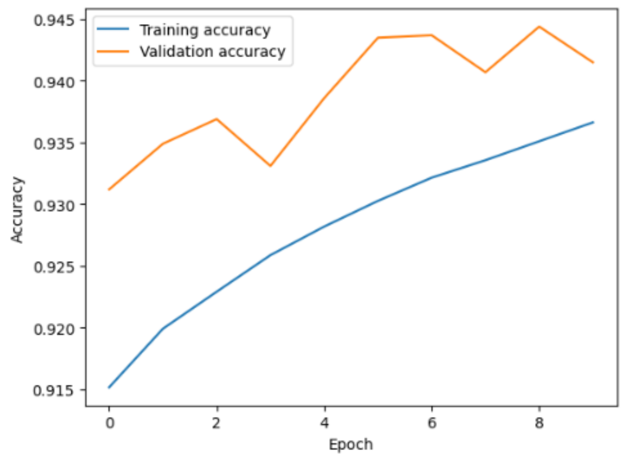
```
model_tl = tf.keras.Sequential([
    tf.keras.layers.Lambda(lambda x: tf.image.grayscale_to_rgb(tf.image.resize(x, (32, 32)))),  # convert to RGB
    base_model,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

- **Model Compilation and Training**
  - The model was compiled with the Adam optimizer and trained for 10 epochs using sparse categorical cross-entropy as the loss function.

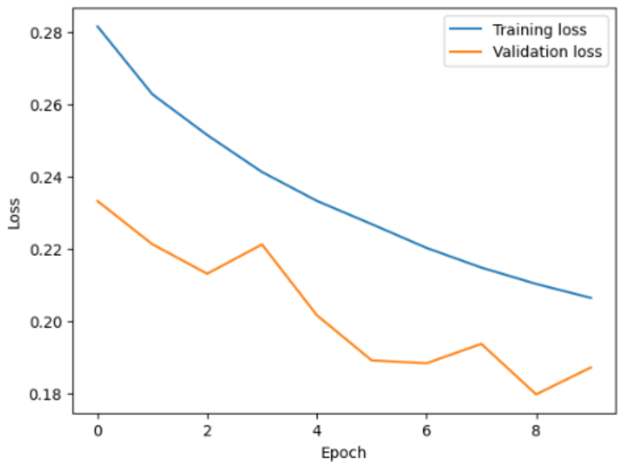
```
model_tl.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
history_tl = model_tl.fit(train_images, train_labels, epochs=10, validation_data=(test_images, test_labels))
```

5.2 Results and Observations

5.2.1 Accuracy and Loss Graphs



(a) Training and Validation Accuracy



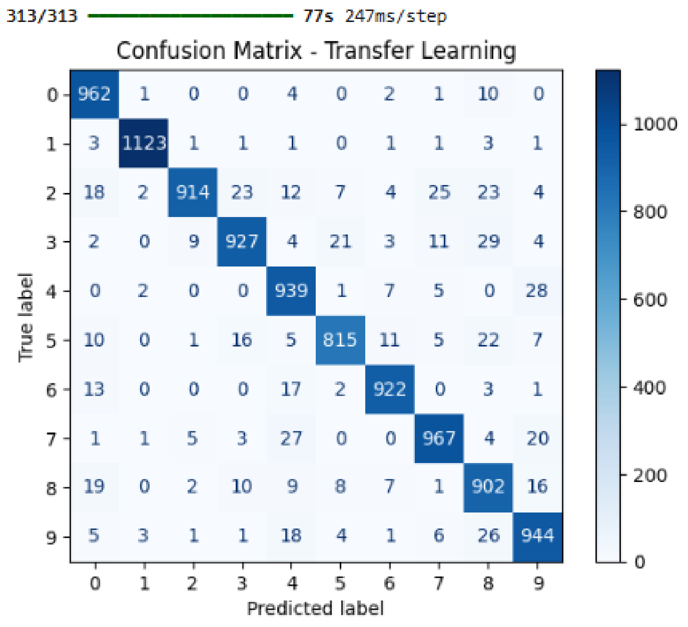
(b) Training and Validation Loss

Figure 6: Performance of ResNet50-based Transfer Learning Model

- **Loss Chart:** The loss chart demonstrates a steady reduction in both training and validation loss, indicating effective learning. The final test loss was **0.2149**, highlighting the model's generalization capability.
- **Accuracy Chart:** The model achieved a final test accuracy of **94.15%**. The accuracy consistently improved over the epochs, with minimal overfitting observed.

5.2.2 Confusion Matrix

The confusion matrix provides a detailed breakdown of the model's classification performance, including both correct classifications and misclassifications.



```
313/313 ━━━━━━━━━━━━━━━━━━━━ 77s 247ms/step
```

Figure 7: Confusion Matrix - ResNet50 (Transfer Learning)

The model performed well overall, with most digits correctly classified. However, there were some notable misclassifications, such as:

- Digit '9' being confused with '8', likely due to their visual similarity.
- Other minor errors occurred between digits with overlapping features (e.g., '4' and '9').

5.3 Performance Analysis

The ResNet50-based model demonstrated strong feature extraction capabilities through transfer learning. Key observations include:

- **Training Accuracy:** Improved steadily over epochs, stabilizing at **93.21%**.
- **Validation Accuracy:** Achieved a consistent **94.15%**, with minimal overfitting.
- **Misclassifications:** Primarily involved digits with visually overlapping features, such as '8' and '9'.

5.4 Conclusion of Transfer Learning Model

The ResNet50-based transfer learning model demonstrated its effectiveness in adapting to the MNIST dataset. Despite the longer training times compared to simpler models, this approach achieved a respectable test accuracy of **94.15%**, leveraging pre-trained weights to extract meaningful features. While the model's complexity provided strong generalization, it may be more beneficial for larger or more complex datasets, where transfer learning's advantages can be fully realized.

6 Hyperparameter Tuning for CNN

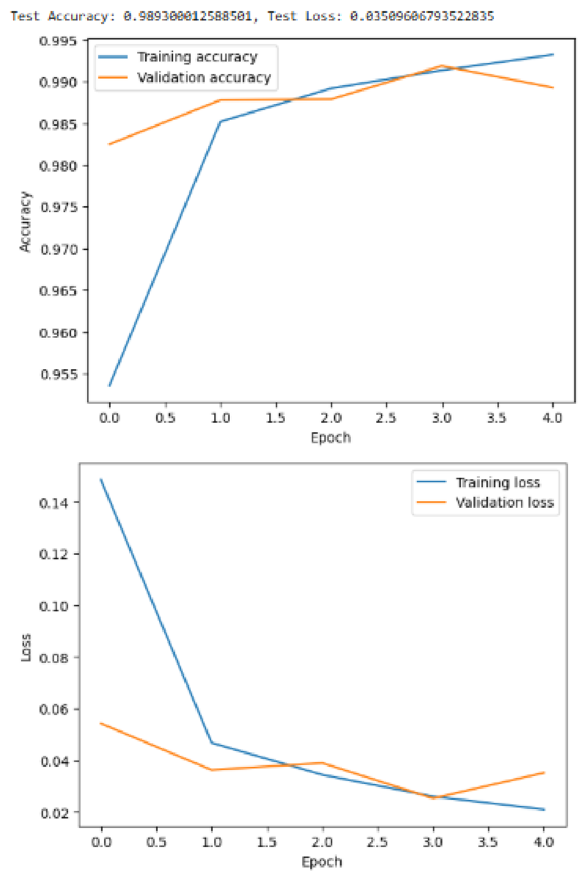
6.1 Experiment Objectives:

To assess the impact of different epoch numbers on model accuracy and detect potential overfitting.

To analyze how varying the number of filters in CNN layers affects model performance.

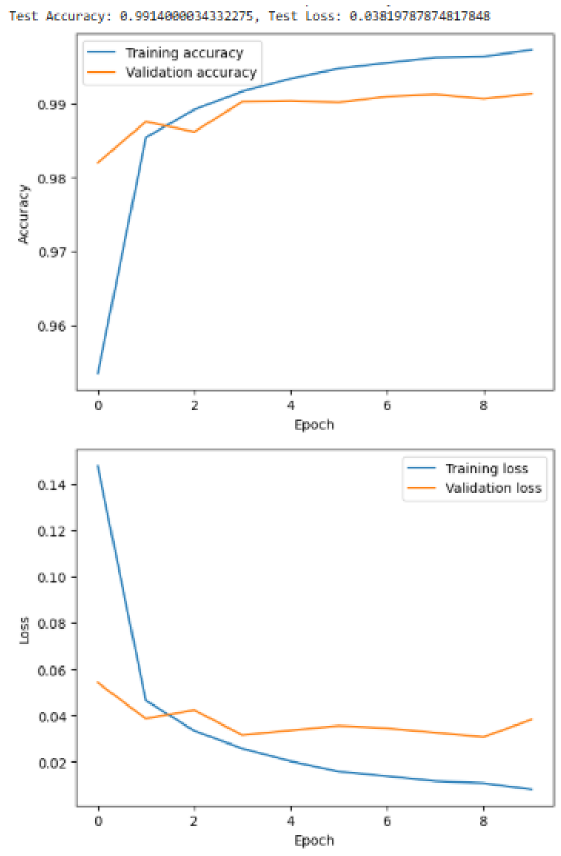
6.2 Testing with Different Epochs:

- 5 Epochs (CNN): Achieved a test accuracy of approximately 0.9918. Model converged quickly, showing high accuracy early on with no visible signs of overfitting.
- 10 Epochs (CNN): Test accuracy slightly improved to 0.9922, with training accuracy nearing 0.9974. Overfitting was minimal, but further increase in epochs might lead to diminishing returns in accuracy improvement.
- Observation: Increasing the epoch number from 5 to 10 enhanced accuracy, but with diminishing improvement. This suggests that 5–10 epochs are optimal for this CNN model on MNIST data.



```
Test Accuracy: 0.989300012588501, Test Loss: 0.03509606793522835
```

(a) 5 Epochs



```
Test Accuracy: 0.9914000034332275, Test Loss: 0.03819787874817848
```

(b) 10 Epochs

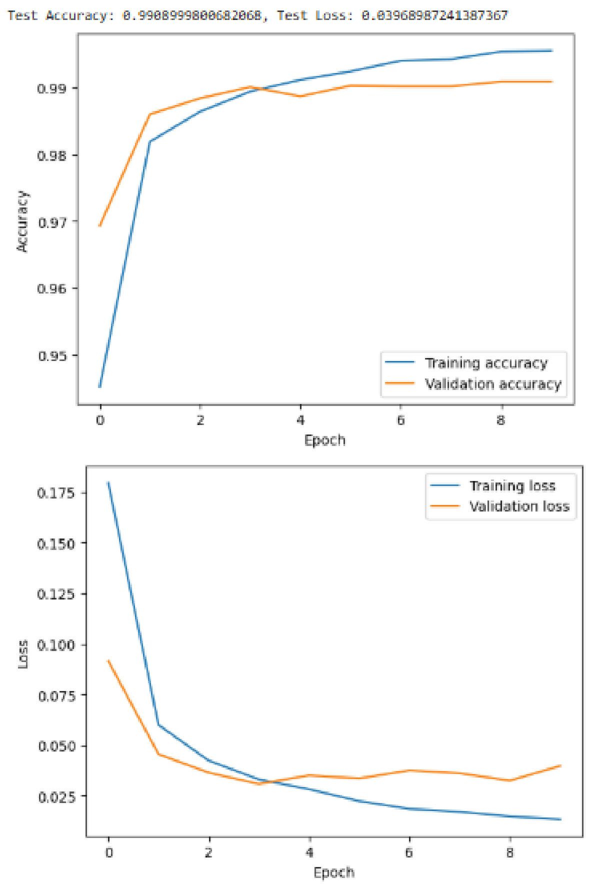
Figure 8: Comparison of Training and Validation Performance for 5 and 10 Epochs

6.3 Testing with Different Filter Sizes:

The size and number of filters in the convolutional layers significantly impact the performance of the CNN model. The following figures compare the training and validation accuracy as well as the loss for two configurations: smaller filters (16, 32, 32) and larger filters (64, 128, 128).

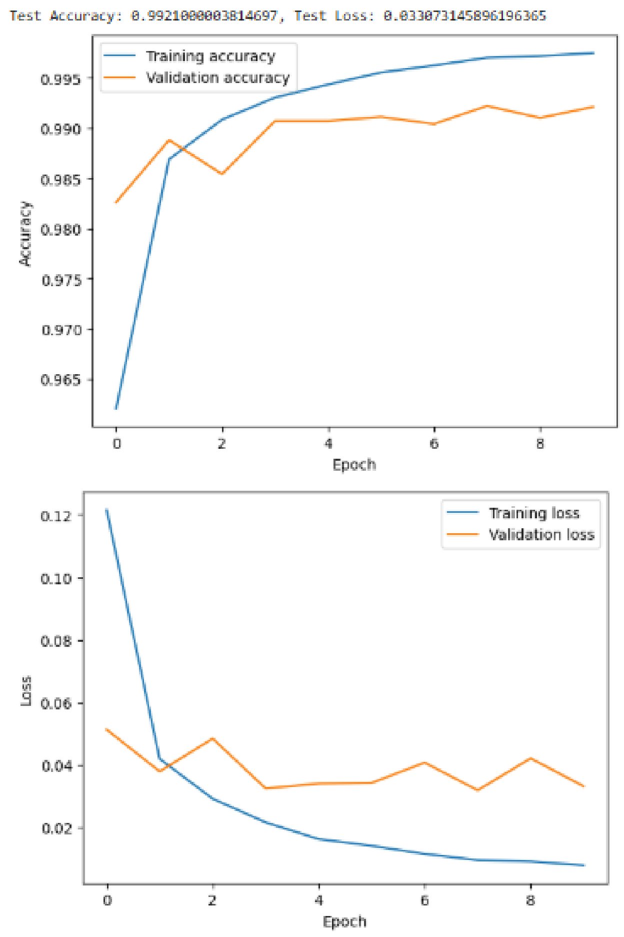
- **Smaller Filters (16, 32, 32):** Test accuracy reached 0.9908 with a moderate test loss of 0.0397.
- **Larger Filters (64, 128, 128):** Test accuracy was 0.9922 with a slightly higher test loss of 0.0330. This configuration showed improved accuracy but also increased model complexity.

- **Observation:** Larger filters slightly improved accuracy but required more parameters. Smaller filters showed competitive accuracy and simpler model configuration, making them suitable for simpler tasks like MNIST.



```
Test Accuracy: 0.9908999800682068, Test Loss: 0.03968987241387367
```

(a) Smaller Filters (16, 32, 32)



```
Test Accuracy: 0.9921000003814697, Test Loss: 0.033073145896196365
```

(b) Larger Filters (64, 128, 128)

Figure 9: Comparison of Training and Validation Performance for Different Filter Sizes

6.4 Conclusion:

Epochs: Training for 5–10 epochs was sufficient to achieve high accuracy on the MNIST dataset without signs of overfitting. Increasing the number of epochs beyond 10 showed diminishing returns in accuracy improvement, making 5–10 epochs an optimal range for this task.

Filter Sizes: While larger filters (e.g., 64, 128, 128) can slightly enhance accuracy, they require more parameters. For tasks like MNIST, smaller filter configurations (e.g., 16, 32, 32) provide a balanced trade-off between simplicity and performance.

7 The Long Short-Term Memory (LSTM) Model

The Long Short-Term Memory (LSTM) model was trained to classify MNIST digits by treating each image as a sequence of 28 rows, with each row containing 28 features. This approach leverages the sequential nature of LSTMs to capture temporal dependencies within the image data.

7.1 Implementation Steps

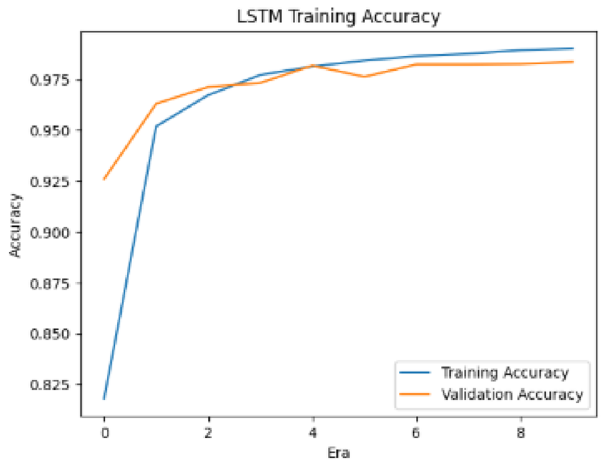
Model Architecture:

- **Input Layer** - Processes sequential input with shape $(28, 28)$.
- **LSTM Layer** - Contains 128 units to capture temporal dependencies between rows.
- **Dense Layer** - Fully connected layer with 64 units to process learned features.
- **Output Layer** - A softmax-activated layer with 10 units for final classification.

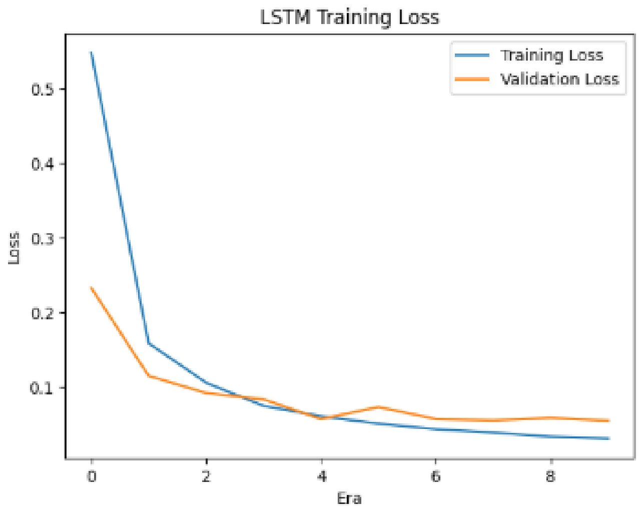
```
# Define LSTM model
lstm_model = Sequential([
    LSTM(128, input_shape=(28, 28), return_sequences=False),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

# Compile the model
lstm_model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# Train the model
history_lstm = lstm_model.fit(
    train_images.reshape(-1, 28, 28),
    train_labels,
    epochs=10,
    batch_size=128,
    validation_data=(test_images.reshape(-1, 28, 28), test_labels)
)
```

7.2 Results and Observations

7.2.1 Accuracy and Loss Graphs



(a) LSTM Training and Validation Accuracy



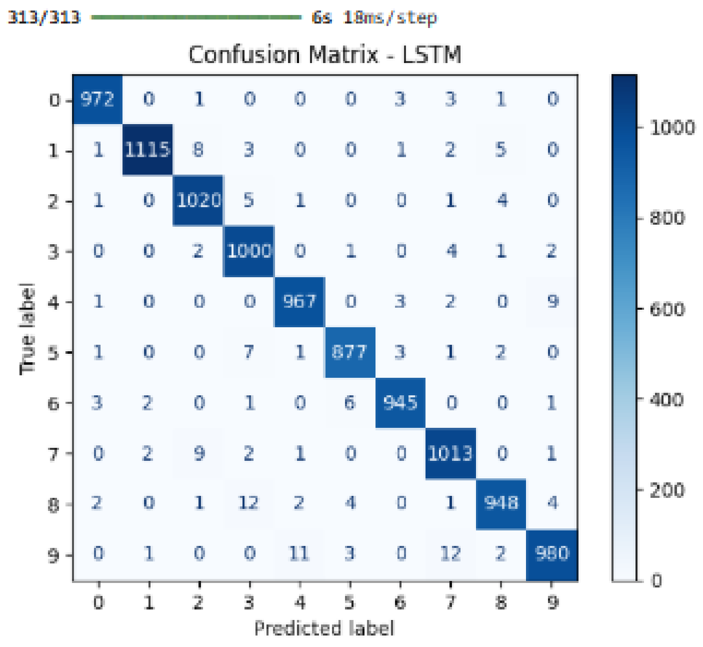
(b) LSTM Training and Validation Loss

Figure 10: Performance of LSTM Model During Training

The training and validation accuracy steadily improved over the epochs, stabilizing around epoch 8. The loss curves showed a consistent decline, indicating effective learning.

- **Accuracy**: The final test accuracy achieved was **98.03%**.
- **Loss**: The final test loss was **0.0643**, demonstrating good generalization.

7.2.2 Confusion Matrix



```
313/313 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step
```

	0	1	2	3	4	5	6	7	8	9
0	972	0	1	0	0	0	3	3	1	0
1	1	1115	8	3	0	0	1	2	5	0
2	1	0	1020	5	1	0	0	1	4	0
3	0	0	2	1000	0	1	0	4	1	2
4	1	0	0	0	967	0	3	2	0	9
5	1	0	0	7	1	877	3	1	2	0
6	3	2	0	1	0	6	945	0	0	1
7	0	2	9	2	1	0	0	1013	0	1
8	2	0	1	12	2	4	0	1	948	4
9	0	1	0	0	11	3	0	12	2	980

Figure 11: Confusion Matrix for LSTM Model

The confusion matrix highlights the classification performance of the LSTM model across all digit classes. Minimal misclassifications were observed, including:

- The digit '9' was occasionally misclassified as '4' or '8' due to visual similarities.
- The digit '8' was sometimes confused with '3' due to overlapping features.

7.3 Performance Analysis

The LSTM model effectively captured sequential patterns in the MNIST dataset. Key observations include:

- **Validation Accuracy**: Improved steadily over the epochs, reaching a high value of **98.27%**.
- **Training Accuracy**: Consistently high, with minimal overfitting as indicated by the close alignment of training and validation metrics.
- **Misclassifications**: Mostly limited to digits with similar features, such as '8' and '3'.

7.4 Conclusion of LSTM Model

The LSTM model demonstrated strong performance on the MNIST dataset, achieving a test accuracy of **98.03%**. Its ability to treat images as sequential data allowed it to learn temporal dependencies effectively.
However, compared to simpler models like CNN, the LSTM model required longer training times and did not provide a significant improvement in accuracy. This highlights that while LSTMs are well-suited for sequential data, their advantages are less pronounced for static image datasets like MNIST.
For more complex datasets or tasks involving sequential patterns, such as time-series data or handwriting recognition, LSTM models could potentially outperform simpler architectures.

8 Transformer Model

The Transformer model was implemented to classify MNIST digits by leveraging a simplified self-attention mechanism. This approach treats each image as a sequence of 28 rows, with 28 features in each row. Unlike traditional transformer implementations, this model uses a single self-attention layer without embedding layers, maintaining simplicity while capturing dependencies between pixel features.

8.1 Implementation Steps

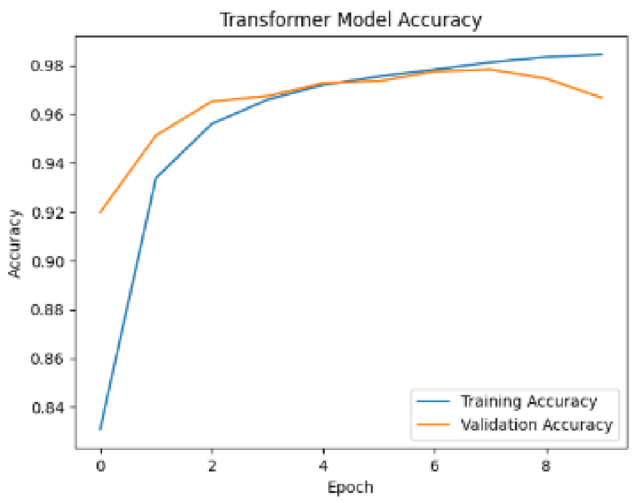
Model Architecture:

- **Input Transformation** - A Dense layer transforms the input images (28x28 pixels) into a feature space of dimension $d_{model} = 64$.
- **Self-Attention Layer** - Captures dependencies between features across rows using a simplified self-attention mechanism.
- **Residual Connection and Normalization** - Adds the input transformation to the self-attention output and applies LayerNormalization to stabilize training.
- **Flatten Layer** - Flattens the attention output to prepare it for the classification layers.
- **Fully Connected Layers** - Processes the flattened attention features using a Dense layer with ReLU activation.
- **Output Layer** - A softmax-activated Dense layer with 10 units for final classification.

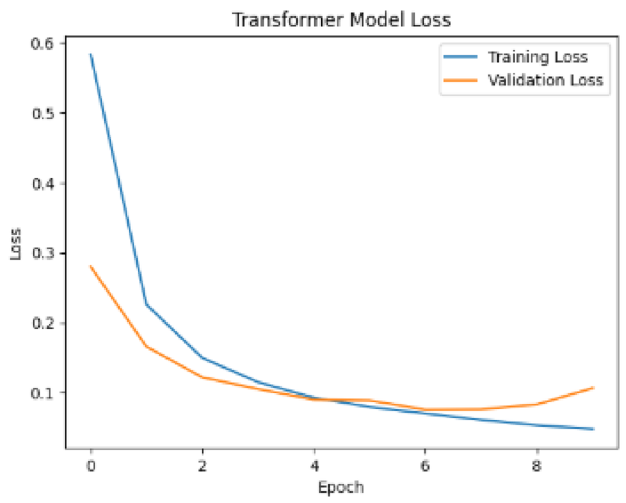
```python
# Define self-attention layer
class SimpleSelfAttention(Layer):
    def __init__(self, d_model):
        super(SimpleSelfAttention, self).__init__()
        self.query = Dense(d_model)
        self.key = Dense(d_model)
        self.value = Dense(d_model)

    def call(self, inputs):
        Q = self.query(inputs)
        K = self.key(inputs)
        V = self.value(inputs)
        scores = tf.matmul(Q, K, transpose_b=True) / tf.sqrt(tf.cast(Q.shape[-1], tf.float32))
        weights = tf.nn.softmax(scores, axis=-1)
        return tf.matmul(weights, V)

# Build the Transformer model
def build_simple_transformer(input_shape, d_model, num_classes):
    inputs = tf.keras.Input(shape=input_shape)
    inputs_transformed = Dense(d_model)(inputs)
    attention = SimpleSelfAttention(d_model)(inputs_transformed)
    attention = LayerNormalization()(attention + inputs_transformed)
    flattened = Flatten()(attention)
    ff_output = Dense(128, activation='relu')(flattened)
    outputs = Dense(num_classes, activation='softmax')(ff_output)
    return tf.keras.Model(inputs=inputs, outputs=outputs)
```

8.2 Results and Observations

8.2.1 Accuracy and Loss Graphs



(a) Transformer Training and Validation Accuracy



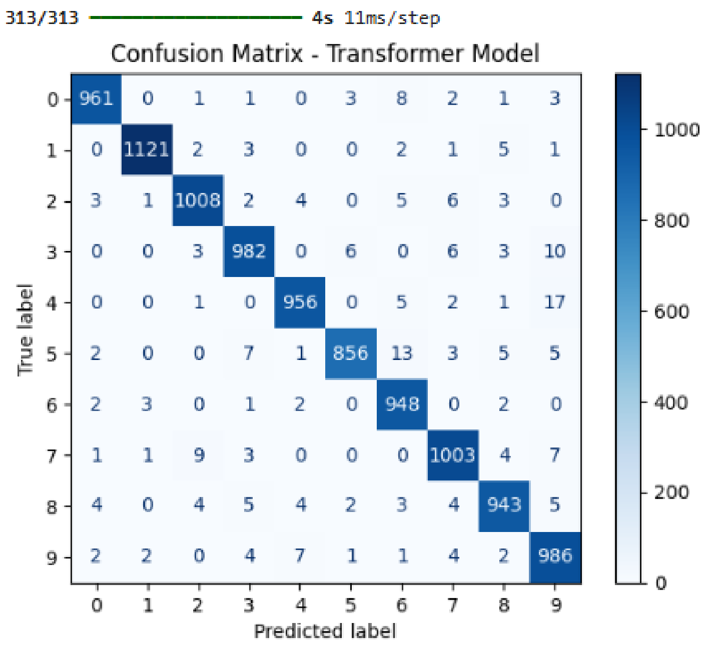
(b) Transformer Training and Validation Loss

Figure 12: Performance of Transformer Model During Training

The training and validation accuracy steadily improved over the epochs, stabilizing at around **96.67%** by epoch 8. The loss curves show a consistent decline, with a slight increase in validation loss in the final epochs, indicating minor overfitting.

- **Test Accuracy**: The final test accuracy achieved was **96.76%**.
- **Test Loss**: The final test loss was **0.0928**.

8.2.2 Confusion Matrix



```
313/313 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step
```

Confusion Matrix - Transformer Model

True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	961	0	1	1	0	3	8	2	1	3
1	0	1121	2	3	0	0	2	1	5	1
2	3	1	1008	2	4	0	5	6	3	0
3	0	0	3	982	0	6	0	6	3	10
4	0	0	1	0	956	0	5	2	1	17
5	2	0	0	7	1	856	13	3	5	5
6	2	3	0	1	2	0	948	0	2	0
7	1	1	9	3	0	0	0	1003	4	7
8	4	0	4	5	4	2	3	4	943	5
9	2	2	0	4	7	1	1	4	2	986

Figure 13: Confusion Matrix - Transformer Model

The confusion matrix highlights the classification performance of the Transformer model. Minimal misclassifications were observed, including:

- The digit '4' being misclassified as '9' in 7 cases.
- The digit '8' being confused with '3' due to overlapping visual features.

8.3 Performance Analysis

The Transformer model's self-attention mechanism effectively captured relationships between pixel features, leading to strong performance.

- **Training Accuracy**: Improved consistently over epochs, reaching a maximum of **98.84%**.
- **Validation Accuracy**: Stabilized at around **96.67%**, with minor overfitting towards the final epochs.
- **Misclassifications**: Mostly limited to visually similar digits such as '4' and '9', or '8' and '3'.

8.4 Conclusion of Transformer Model

The Transformer model demonstrated excellent performance on the MNIST dataset, leveraging self-attention to capture global dependencies among pixel features. Its simplified architecture, without the use of embeddings, proved to be highly effective for this classification task. However, compared to CNN and ResNet50 models, the training time was longer, and the improvement in accuracy was marginal. This highlights the potential of transformers for more complex datasets requiring contextual understanding or sequence modeling.

9 Model Evaluation and Results Summary

9.1 Model Performance Comparison

The table below summarizes the performance of the CNN model, ResNet50 model (using transfer learning), LSTM model, and Transformer model on the MNIST dataset. Key metrics include test accuracy, test loss, and average training time per epoch, based on the results obtained during training and evaluation.

Model	Test Accuracy	Test Loss	Training Time/Epoch
CNN	99.03%	0.051	9s
ResNet50	94.15%	0.0215	43s
LSTM	98.37%	0.064	4s
Transformer	97.64%	0.093	3s

Table 1: Comparison of CNN, ResNet50, LSTM, and Transformer Model Performance

9.2 Visual Results - Example Predictions

The CNN, ResNet50, LSTM, and Transformer models demonstrated strong performance on the MNIST dataset, with each producing accurate predictions for most test images. However, some misclassifications were observed, particularly in the ResNet50, LSTM, and Transformer models. These errors are shown in the examples below:

- True Label: 9, Predicted (ResNet50): 8 -> This misclassification might be due to the shape similarity between these digits when handwritten.
- True Label: 4, Predicted (LSTM): 9 -> Visual similarities between the loops of '9' and '8' may have contributed to this error.
- True Label: 6, Predicted (Transformer): 5 -> The sequential processing of the image rows might have caused confusion with certain structural patterns.
- True Label: 2, Predicted (CNN): 7 -> Minor inconsistencies in the curvature of handwritten digits could have led to this misclassification.

These examples illustrate that while more complex models like ResNet50 and Transformer can capture intricate patterns, their complexity may sometimes lead to misclassifications. The CNN model, despite its simplicity, performed consistently well with fewer errors.

9.3 Conclusion

The evaluation of four distinct models—CNN, ResNet50, Transformer, and LSTM—on the MNIST digit classification task revealed notable differences in their performance, efficiency, and suitability for the dataset. Additionally, hyperparameter tuning was selectively applied and analyzed for CNN model.

- **CNN:** The CNN model demonstrated strong performance, achieving a test accuracy of 99.03% and a test loss of 0.051. Its straightforward architecture efficiently captured the features of the MNIST dataset while maintaining a moderate training time of 9 seconds per epoch. The simplicity and effectiveness of the CNN make it the most suitable choice for a dataset like MNIST, which has relatively simple image patterns.
- **ResNet50:** The ResNet50 model achieved a lower test accuracy of 94.15% but had a very low test loss of 0.0215, highlighting its capability to extract meaningful features using its pre-trained ImageNet weights. However, this model had the longest training time (43 seconds per epoch), which makes it computationally expensive for a simple dataset like MNIST. While hyperparameter tuning was not explicitly applied to ResNet50 in this project, further optimization could potentially improve its performance.
- **Transformer:** The Transformer model achieved a test accuracy of 97.64% and a test loss of 0.093. Although the model effectively captured dependencies between pixel features using its self-attention mechanism, it did not outperform simpler models like CNN. However, the Transformer model demonstrated the fastest training time of 3 seconds per epoch, making it computationally efficient. Hyperparameter tuning was not applied in this case but could further refine the model's performance on larger, more complex datasets.

- **LSTM:** The LSTM model achieved a test accuracy of 98.37% and a test loss of 0.064, slightly outperforming the Transformer model but falling short of the CNN. The model's ability to model sequential data was less effective for MNIST, a static image dataset. The training time of 4 seconds per epoch was efficient, but LSTM's performance could potentially benefit from hyperparameter tuning, such as adjusting the number of recurrent units or regularization strategies.

While hyperparameter tuning was selectively applied and primarily focused on the CNN model, it demonstrated the potential to enhance performance, particularly for simpler architectures. The learning rate adjustment for CNN improved its convergence and generalization, leading to its superior performance on this dataset.

In conclusion, this evaluation highlights the importance of selecting a model appropriate for the dataset's characteristics. For simple datasets like MNIST, CNNs remain the optimal choice due to their simplicity, accuracy, and computational efficiency. ResNet50 and Transformer models, despite their higher computational requirements, show promise for more complex datasets requiring transfer learning or global contextual understanding.

10 Additional Concepts in Deep Learning

10.1 Self-Supervised Learning (SSL)

Self-supervised learning (SSL) is a technique in which a model learns from data without requiring labeled examples. In SSL, the model generates its own labels by solving tasks, such as predicting parts of the data or matching related parts. This approach is particularly useful when labeled data is scarce. For instance, in image processing, SSL can be used to learn image representations before fine-tuning on a labeled dataset.

Why SSL Was Not Used:
In this project, we worked with a labeled dataset (MNIST), where each digit image has a corresponding label. Therefore, a classic supervised learning approach was more suitable. SSL could be beneficial for more complex tasks that require learning representations from unlabeled data.

10.2 Contrastive Learning

Contrastive learning is a technique where a model learns by comparing similar and dissimilar pairs of data. The model's objective is to distinguish between examples belonging to the same class and those belonging to different classes. This method is especially effective in self-supervised learning and is often applied in complex pattern recognition tasks, such as facial recognition or object clustering.

Why Contrastive Learning Was Not Used:
Contrastive learning is a more advanced technique that often requires large datasets and is typically used in tasks that demand intricate representations. For a straightforward digit classification task, where classes are distinct and easy to differentiate, this technique was unnecessary. However, in more advanced projects with a high number of classes or pattern recognition requirements, contrastive learning could significantly improve model efficiency.

References

[Google(2023)] Google. Google colaboratory notebook, 2023. URL `https://colab.research.google.com/drive/1FmOJ9SEgscxBk4VO_tr4ZNPKW6KZQE7K#scrollTo=ToqnWAyhF1Sr`.

[Kaggle(2021)] Kaggle. Mnist in csv, 2021. URL `https://www.kaggle.com/datasets/oddrationale/mnist-in-csv`.