Task 2: MNIST Digit Classification with Convolutional Neural Networks

1. Introduction

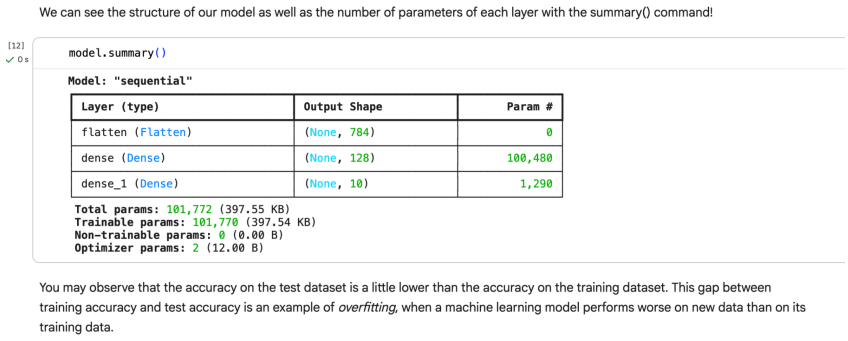
This report details the development and evaluation of neural network models for classifying handwritten digits from the MNIST dataset. The primary objective was to compare the performance of a simple Fully Connected (FC) neural network against a more complex Convolutional Neural Network (CNN) and to demonstrate an understanding of the underlying training mechanics. The MNIST dataset, comprising 70,000 28x28 pixel grayscale images, was used for training and evaluation.

2. Fully Connected Neural Network (Baseline Model)

A baseline model was first implemented using a simple Fully Connected architecture. This model served as a benchmark to highlight the performance improvements offered by CNNs.

2.1. Model Architecture

The FC model consisted of a Flatten layer to transform the 28x28 images into a 1D vector, followed by a Dense hidden layer with 128 units and ReLU activation, and a final Dense output layer with 10 units and a Softmax activation to produce a probability distribution over the 10 digit classes.



We can see the structure of our model as well as the number of parameters of each layer with the summary() command!

[12] ✓ 0s

```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 128)	100,480
dense_1 (Dense)	(None, 10)	1,290

Total params: 101,772 (397.55 KB)
Trainable params: 101,770 (397.54 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 2 (12.00 B)

You may observe that the accuracy on the test dataset is a little lower than the accuracy on the training dataset. This gap between training accuracy and test accuracy is an example of *overfitting*, when a machine learning model performs worse on new data than on its training data.

Figure 1: Architecture summary of the baseline Fully Connected model, showing 101,770 trainable parameters.

2.2. Training and Evaluation

The model was compiled with a Stochastic Gradient Descent (SGD) optimizer and trained for 5 epochs. It achieved a high training accuracy, but the critical metric is its performance on the unseen test set.

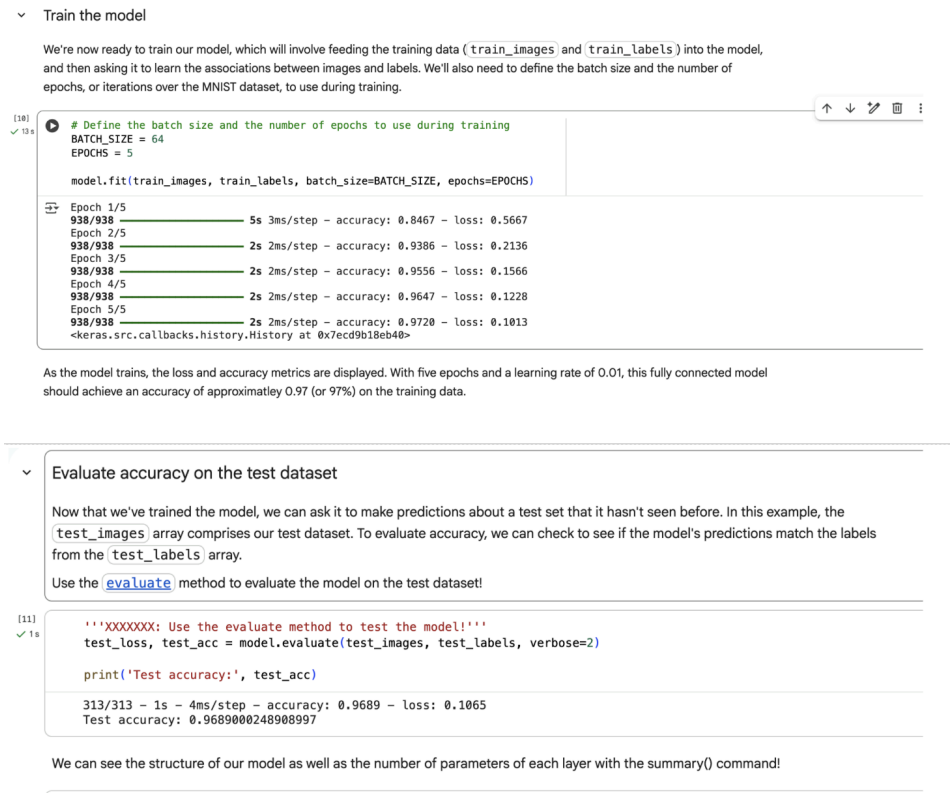


Figure 2: Training logs and final evaluation of the FC model, showing a test accuracy of 96.89%.

2.3 Observation on Overfitting

The model achieved a test accuracy of 96.89%, which is slightly lower than its training accuracy. This small gap is a classic indicator of overfitting, where the model learns the training data too well and its performance slightly degrades on new, unseen data.

3. Convolutional Neural Network (CNN)

To improve upon the baseline, a Convolutional Neural Network was designed. CNNs are inherently better suited for image data due to their ability to capture spatial hierarchies and patterns through convolutional and pooling layers.

3.1 Model Architecture

The CNN model was significantly more complex, featuring three blocks of Conv2D and MaxPooling2D layers, followed by two Dense layers. This architecture allows the network to learn features hierarchically, from simple edges to complex shapes.

Define the CNN model

```
def build_cnn_model():
    cnn_model = tf.keras.Sequential([

        # XXXXXXX: Define the first convolutional layer
        tf.keras.layers.Conv2D(64, (3,3), activation='relu', input_shape=(28,28,1)),

        # XXXXXXX: Define the first max pooling layer
        tf.keras.layers.MaxPool2D((2,2)),

        # XXXXXXX: Define the second convolutional layer
        tf.keras.layers.Conv2D(128, (3,3), activation='relu'),

        # XXXXXXX: Define the second max pooling layer
        tf.keras.layers.MaxPool2D((2,2)),

        tf.keras.layers.Conv2D(256, (3,3), activation='relu'),
        tf.keras.layers.MaxPool2D((2,2)),

        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(256, activation=tf.nn.relu),


        # XXXXXXX: Define the last Dense layer to output the classification
        # probabilities. Pay attention to the activation needed a probability
        # output
        tf.keras.layers.Dense(10, activation='softmax')
    ])

    return cnn_model

cnn_model = build_cnn_model()
# Initialize the model by passing some data through
cnn_model.predict(train_images[[0]])
# Print the summary of the layers in the model.
print(cnn_model.summary())
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/convolutional/base_conv.py:113:
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 64)	640
max_pooling2d (MaxPooling2D)	(None, 13, 13, 64)	0
conv2d_1 (Conv2D)	(None, 11, 11, 128)	73,856
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 128)	0
conv2d_2 (Conv2D)	(None, 3, 3, 256)	295,168
max_pooling2d_2 (MaxPooling2D)	(None, 1, 1, 256)	0
flatten_1 (Flatten)	(None, 256)	0
dense_2 (Dense)	(None, 256)	65,792
dense_3 (Dense)	(None, 10)	2,570

```
Total params: 438,026 (1.67 MB)
Trainable params: 438,026 (1.67 MB)
Non-trainable params: 0 (0.00 B)
None
```

Figure 3: Architecture summary of the CNN model, showing 438,026 parameters and the progressive downsampling of the image dimensions.

3.2 Training and Evaluation

The CNN was compiled with the Adam optimizer and trained for 5 epochs. It learned more efficiently and effectively than the FC model.

Train and test the CNN model

Now, as before, we can define the loss function, optimizer, and metrics through the `compile` method. Compile the CNN model with an optimizer and learning rate of choice:

```
'''XXXXXXX: Define the compile operation with your optimizer and learning rate of choice'''
cnn_model.compile(optimizer=tf.keras.optimizers.Adam(), loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

As was the case with the fully connected model, we can train our CNN using the `fit` method via the Keras API.

```
'''XXXXXXX: Use model.fit to train the CNN model, with the same batch_size and number of epochs previously used.'''
cnn_model.fit(train_images, train_labels, batch_size=BATCH_SIZE, epochs=EPOCHS)
```

```
Epoch 1/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.8680 - loss: 0.4073
Epoch 2/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - accuracy: 0.9821 - loss: 0.0574
Epoch 3/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.9871 - loss: 0.0401
Epoch 4/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.9910 - loss: 0.0294
Epoch 5/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - accuracy: 0.9932 - loss: 0.0204
<keras.src.callbacks.history.History at 0x7ecd9031aed0>
```

Great! Now that we've trained the model, let's evaluate it on the test dataset using the `evaluate` method:

```
'''XXXXXXX: Use the evaluate method to test the model!'''
test_loss, test_acc = cnn_model.evaluate(test_images, test_labels, verbose=2)

print('Test accuracy:', test_acc)
```

```
313/313 - 2s - 7ms/step - accuracy: 0.9855 - loss: 0.0489
Test accuracy: 0.9854999780654907
```

Make predictions with the CNN model

With the model trained, we can use it to make predictions about some images. The `predict` function call generates the output predictions given a set of input samples.

```
predictions = cnn_model.predict(test_images)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
```

With this function call, the model has predicted the label for each image in the testing set. Let's take a look at the prediction for the first image in the test dataset:

```
predictions[0]
```

```
array([3.4556813e-08, 1.7851081e-06, 3.2553742e-06, 1.0358535e-06,
       3.3359896e-07, 7.6598168e-09, 9.0799368e-11, 9.9999142e-01,
       2.8100919e-08, 2.1139410e-06], dtype=float32)
```

As you can see, a prediction is an array of 10 numbers. Recall that the output of our model is a probability distribution over the 10 digit classes. Thus, these numbers describe the model's "confidence" that the image corresponds to each of the 10 different digits.

Let's look at the digit that has the highest confidence for the first image in the test dataset:

```
'''XXXXXXX: identify the digit with the highest confidence prediction for the first
    image in the test dataset. '''
prediction = np.argmax(predictions[0])

print(prediction)
```

```
7
```

Figure 4: Training logs and final evaluation of the CNN model, showing a superior test accuracy of 98.55%.

3.3 Key Result and Comparison

The CNN model achieved a test accuracy of 98.55%, which is a significant improvement of 1.66% over the FC baseline. Furthermore, the gap between training and test accuracy was smaller, indicating that the CNN generalizes better and is less prone to overfitting. This conclusively demonstrates the superiority of CNNs for image classification tasks.

4. Model Predictions and Analysis

To qualitatively assess the CNN's performance, its predictions were visualized on a sample of test images.

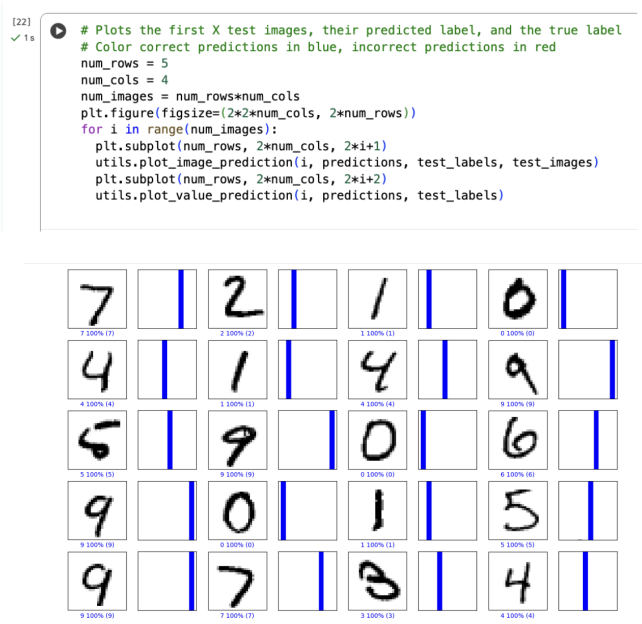


Figure 5: A grid of test images showing the model's predictions. Correct predictions are in blue. The model achieved 100% accuracy on this sample, with most predictions showing 100% confidence.

The visualization confirms the model's high accuracy and shows that it is often extremely confident in its correct predictions, which aligns with the quantitative results from the evaluation.

5. Custom Training Loop with GradientTape

To deepen the understanding of the training process, a custom training loop was implemented using TensorFlow's tf.GradientTape. This method provides low-level control over the steps of forward pass, loss calculation, gradient computation, and parameter updates.

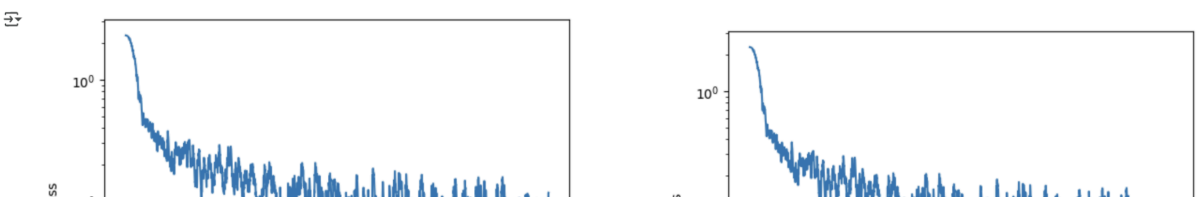


Figure 6: The loss curve from the custom training loop, showing a consistent decrease in loss over iterations, confirming the successful implementation of backpropagation.

The steadily decreasing loss curve validates that the backpropagation algorithm was correctly implemented, demonstrating a fundamental understanding of how neural networks learn.

6. Conclusion

This laboratory successfully demonstrated the complete workflow for an image classification project. The key conclusions are:

- Architectural Superiority: The Convolutional Neural Network (98.55% test accuracy) conclusively outperformed the Fully Connected Network (96.89%), validating that CNNs are fundamentally better for image-related tasks.
- Improved Generalization: The CNN exhibited less overfitting, shown by a smaller gap between its training and test performance, highlighting its robustness and better generalization capabilities.
- High Confidence and Accuracy: Qualitative analysis confirmed the model's high accuracy and its strong confidence in correct predictions.
- Fundamental Understanding: The successful implementation of a custom training loop with tf.GradientTape provided valuable insight into the core mechanics of neural network training.

In summary, the transition from a simple FC network to a CNN resulted in a more accurate, robust, and well-performing model for the MNIST digit classification task.