

Practical Benchmarking and Hands-On Evaluation of AI Popular Systems is a companion volume to AI Popular Systems: Architecture, Capabilities, and Comparative Analysis. While the first book examines the design, features, integration, and broader characteristics of major AI platforms, this volume evaluates their practical performance through a shared experimental framework. Using CNN development on the MNIST dataset as a common task, the book compares systems such as ChatGPT, Claude, Gemini, Copilot, Meta Llama, Gemma, Mistral, Hugging Chat, and Zapier. It examines prompt effectiveness, code generation, debugging support, model design, training, visualization, optimization, and final accuracy. The volume combines reproducible experiments, code examples, figures, and comparative observations to show how modern AI assistants perform in realistic machine learning workflows. It is intended for students, researchers, educators, AI engineers, and practitioners interested in applied evaluation of generative AI systems.



Alexander I. Iliev

Practical Benchmarking and Hands-On Evaluation of AI Popular Systems

A Comparative Experimental Guide to Modern AI Platforms



Prof. Dr. Alexander I. Iliev is a professor and researcher in Artificial Intelligence, Machine Learning, Big Data Analytics, and Smart Systems. He holds three patents and has authored numerous scientific publications and books. He is affiliated with UC Berkeley, SRH University of Applied Sciences Heidelberg, and the Bulgarian Academy of Sciences.



9 786630 225143

Alexander I. Iliev

**Practical Benchmarking and Hands-On Evaluation of AI Popular
Systems**

FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY

Alexander I. Iliev

Practical Benchmarking and Hands-On Evaluation of AI Popular Systems

**A Comparative Experimental Guide to Modern AI
Platforms**

LAP LAMBERT Academic Publishing

Imprint

Any brand names and product names mentioned in this book are subject to trademark, brand or patent protection and are trademarks or registered trademarks of their respective holders. The use of brand names, product names, common names, trade names, product descriptions etc. even without a particular marking in this work is in no way to be construed to mean that such names may be regarded as unrestricted in respect of trademark and brand protection legislation and could thus be used by anyone.

Cover image: www.ingimage.com

Publisher:

LAP LAMBERT Academic Publishing

is a trademark of

Dodo Books Indian Ocean Ltd. and OmniScriptum S.R.L publishing group

120 High Road, East Finchley, London, N2 9ED, United Kingdom

Str. Armeneasca 28/1, office 1, Chisinau MD-2012, Republic of Moldova,
Europe

Managing Directors: Ieva Konstantinova, Victoria Ursu

info@omniscryptum.com

Printed at: see last page

ISBN: 978-66-30-22514-3

Copyright © Alexander I. Iliev

Copyright © 2026 Dodo Books Indian Ocean Ltd. and OmniScriptum S.R.L
publishing group

FOR AUTHOR USE ONLY

Practical **B**enchmarking and Hands-**O**n **E**valuation of **A**I Popular **S**ystems

*A **C**omparative **E**xperimental **G**uide to **M**odern
AI **P**latforms*

Author:

Prof Dr Alexander Iliev Iliev

Scientific Editors:

Prof Dr Reiner Creutzburg

SRH University of Applied Sciences, Berlin

Prof Dr Daniel Arias Aranda

University of Granada, Granada

Scientific Reviewers:

Prof Dr Goran Rafajlovski

SRH University of Applied Sciences, Berlin

Prof Dr Vladimir Stanchev

SRH University of Applied Sciences, Berlin

A Note from the Author:

This manuscript was written to demonstrate how today's leading AI systems can be evaluated through practical experimentation rather than theoretical comparison alone. By applying a common benchmarking methodology to identical machine learning tasks, the book provides readers with direct insight into how different AI platforms generate code, solve engineering problems, assist with debugging, and support the complete development workflow. The emphasis is on practical experience, reproducibility, and objective comparison within a real-world AI engineering environment.

This work complements ***AI Popular Systems: Architecture, Capabilities, and Comparative Analysis***, which presents the architecture, capabilities, integration, ethics, and ecosystem of the AI systems discussed here. Together, the two volumes present a balanced perspective on modern artificial intelligence by combining architectural analysis with practical experimentation, enabling readers to understand both how AI systems are designed and how they perform in real-world engineering tasks.

My expectation is that this work serves as a comprehensive guide for researchers, students, AI practitioners, software architects, and anyone interested in understanding not only what today's AI systems are capable of, but also how they perform when applied to practical machine learning tasks.

Prof. Dr. Alexander Iliev Iliev

Table Of Contents

Practical Benchmarking CNN/MNIST Experiments Across Popular AI systems	7
Chapter 1 - Design a CNN Network Using the MNIST Dataset with ChatGPT 3.5.....	7
1.1. CNN Network designed using MNIST Dataset:	7
1.2. Graph Plotted by Using the Code Given by ChatGPT 3.5.....	14
1.3. Experience in using ChatGPT 3.5 to Design CNN Network Using MNIST Dataset:	14
Chapter 2 - Designing a CNN Network using MNIST Dataset with ChatGPT4.....	16
2.1. Introduction	16
2.2. Model Development Process.....	16
2.3 Results.....	21
2.4. Plot Training and Validation Metrics	23
2.5. Confusion Matrix:.....	23
2.6. ROC Curve and AUC (for each class)	24
2.7. Precision-Recall Curve:	24
2.8. Conclusion	26
Chapter 3 - Design a CNN Network Using the MNIST Dataset with ChatGPT 5.2.....	27
3.1. CNN Network designed using MNIST Dataset:	27
3.2. Graphs Plotted by Using the Code Given by ChatGPT 5.2	35
3.3. Experience in using ChatGPT 5.2 to Design CNN Network Using MNIST	39
3.4. Conclusion	41
Chapter 4 - Designing a CNN Network using the MNIST Dataset by using Claude ..	43
4.1. Introduction	43
4.2. Architecture, Model, Methods, Formulas, design	43
4.3. Results.....	52
4.4. Conclusion	55
Chapter 5 - Designing a CNN Network using the MNIST Dataset with Gemini AI	56
5.1. Prompt 1 <i>Design (CNN) model for classifying handwritten digits</i>	56
5.2. Prompt 2 <i>Design a CNN network and visualize the predictions</i>	57
5.3. Prompt 3 <i>Implement a hyper parameter tuning on a CNN model</i>	58
5.4. Prompt 4 <i>Use the latest version of TensorFlow to create a CNN network</i>	59
5.5. Prompt 5 <i>Perform cross validation and give me the best tuning parameters.</i>	60
5.6. Prompt 6 <i>Perform cross validation on CNN model using Adam optimizer</i>	60

5.7. Prompt 7 <i>This is outdated. Provide a compatible solution with latest Keras</i>	60
5.8. Prompt 8 <i>Now prioritize time efficiency and provide code with max 3 epochs.</i>	61
5.9. Prompt 9 <i>In this above change some values try with different methods and run it.</i>	61
5.10. Prompt 10 <i>Plot the changes in accuracy over time.</i>	62
5.11. Prompt 11 <i>Design the CNN model using Keras and MNIST data and tune it up</i> ..	62
5.12. Prompt 12 <i>Optimize the results using k-fold cross validation.</i>	63
5.13. Prompt 13 <i>Change the objective to Precision</i>	65
5.14. Prompt 14 <i>Design the CNN model - try different layers and plot accuracy.</i>	65
5.15. Prompt 15 <i>Create a CNN model - visualization based on feature maps</i>	67
5.16. Prompt 16 <i>Design the CNN model - change the activation functions</i>	68
5.17. Prompt 17 <i>Design the CNN model - tune the learning rate</i>	69
5.18. Prompt 18 <i>HyperModel class for KerasTuner</i>	70
Chapter 6 - CNN for MNIST using Zapier Chatbot	71
6.1. Introduction	71
6.2. Data Loading	71
6.3. Data Processing	72
6.4. Data Splitting	72
6.5. Model Evaluation	81
6.6. Conclusion	83
Chapter 7 - CNN for MNIST using Bing AI	84
7.1. MNIST Dataset	84
7.2. Literature Review	84
7.3. Objective	84
7.4. Methodology	85
7.5. Data Preprocessing	85
7.6. Model Architecture	86
7.7. Training and Evaluation	87
7.8. Results	92
7.9. Insights and Observations	95
7.8. Conclusion	96
Chapter 8 - CNN for MNIST using Meta Llama	97
8.1. Evaluation of Meta Llama 2 Code Generation Capability Using CNN Accuracy on MNIST Dataset	97
8.2. Methodology	97
8.3. Results	99

8.4. Conclusion	100
8.5. Experiments	101
Chapter 9 - Building a CNN Network using Gemma AI.....	111
9.1. Literature Review	111
9.2. Methodology	112
9.3. Implementation	113
9.4. Setting Up the Environment:	115
9.5. Designing and training the CNN Model:	115
9.6. Prompts and Outputs analysis:	116
9.7. Results	118
9.8. Model Performance:	118
9.9. Conclusion	120
Chapter 10 - Design a CNN Network Using the MNIST Dataset with Mistral:.....	122
10.1. Objective	122
10.2. Methodology	122
10.3. CNN Network designed using MNIST Dataset:	122
10.4. Importing libraries	123
10.5. Data Loading and Preprocessing	124
10.6. Model Definition	124
10.7. Model Compilation	127
10.8. Model Training	127
10.9. Model Evaluation	128
10.10. Results	128
Accuracy Graph:	128
Chapter 11 - Design CNN Network using MNIST Dataset with Hugging Chat.....	131
11.1. Introduction	131
11.2. Literature Review	133
11.3. Methodology	135
11.4. Architecture.....	137
Convolutional Layers:	137
11.5. Methods:	138
11.6. Results:	139
11.7. Interpretation:	143
11.8. Discussion:	145

11.9. Conclusion: 148

11.10. Future work and directions: 150

References.....152

FOR AUTHOR USE ONLY

Practical Benchmarking

CNN/MNIST Experiments Across Popular AI systems

Chapter 1 - Design a CNN Network Using the MNIST Dataset with ChatGPT 3.5

In this research, we will use the MNIST dataset as a guide and use ChatGPT 3.5 to assist us in creating a Convolutional Neural Network (CNN). Designing a Convolutional Neural Network (CNN) to classify digits from the MNIST dataset involves a structured process to ensure the model is both effective and efficient. Utilizing Keras, a high-level API in TensorFlow (Abadi et al., 2016), we can systematically approach this task through several steps. First, we import the necessary libraries to set up our environment. Then, we load the MNIST dataset and preprocess the data to make it suitable for training. Next, we define the architecture of our CNN model, specifying the layers and their configurations. Following the model definition, we compiled it, setting the loss function, optimizer, and metrics for evaluation. Once compiled, we train the model using the training data, allowing it to learn and optimize. Finally, we evaluate the model's performance using test data to ensure it generalizes well to unseen data. This step-by-step implementation provides a comprehensive approach to designing and training a CNN using the MNIST dataset.

1.1. CNN Network designed using MNIST Dataset:

A Step-by-Step Guide Using ChatGPT 3.5

This is a simple Convolutional Neural Network (CNN) implementation that uses the TensorFlow Keras module to categorize numbers from the MNIST dataset.

Step-by-Step Implementation:

Step 1 - Import Libraries

Step 2 - Load and Preprocess Data

Step 3 - Define the CNN Model

Step 4 - Compile the Model

Step 5 - Train the Model

Step 6 - Evaluate the Model

Step 1: Import Libraries:

First, we will import the necessary libraries. TensorFlow provides high-level APIs for building and training models, while Matplotlib is used for visualizing the data.

```
pip install tensorflow matplotlib
```

Code Snippet 1.1. Install Matplotlib Libraries.

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
```

Code Snippet 1.2. Import TensorFlow.

In this initial step, Chat GPT 3.5 imports the necessary libraries for building and training a Convolutional Neural Network (CNN). Chat GPT 3.5 uses TensorFlow and Keras, which are powerful libraries for machine learning and deep learning.

Specifically, Chat GPT 3.5 imports TensorFlow and its submodules Keras. datasets for loading datasets, Keras's layers for building the model's layers, and Keras models for creating the model.

Additionally, Chat GPT 3.5 imports matplotlib. Py plot to visualize the data and training results.

By printing the TensorFlow version and the number of available GPUs, Chat GPT 3.5 ensures our environment is set up correctly for efficient computation.

```
import tensorflow as tf
print(tf.__version__)

2.16.1
```

Code Snippet 1.3. Printing the TensorFlow Version.

Step 2: Load and Preprocess Data:

```
# Load the dataset
(train_images, train_labels), (test_images, test_labels) = datasets.mnist.load_data()

# Normalize pixel values to be between 0 and 1
train_images = train_images / 255.0
test_images = test_images / 255.0

# Reshape images to include the channel dimension (1 in this case for grayscale)
train_images = train_images.reshape((train_images.shape[0], 28, 28, 1))
test_images = test_images.reshape((test_images.shape[0], 28, 28, 1))
```

Code Snippet 1.4. Loading the MNIST Dataset.

The MNIST dataset is a well-known benchmark in the field of machine learning and computer vision, specifically designed for handwritten digit classification. It comprises a large collection of grayscale images, each depicting a single handwritten digit ranging from 0 to 9. The dataset is divided into two parts: 60,000 training images, which are used to train machine learning models, and 10,000 test images, which are used to evaluate the performance of these models. The task is to accurately classify these images into one of the ten-digit categories. TensorFlow, a popular deep learning framework, provides a built-in function to conveniently load this dataset, simplifying the process of getting started with digit classification tasks.

Once the dataset is loaded, the pixel values, originally ranging from 0 to 255, are normalized to a range between 0 and 1. This normalization step is crucial

as it improves the performance and convergence speed of neural networks by ensuring that the input values are within a consistent range. Additionally, the images are reshaped to include a channel dimension, converting them from a 3D array (28x28 pixels) to a 4D array (28x28x1). This reshaping is necessary because convolutional neural networks (CNNs) require a specific input shape that includes the number of channels, even for grayscale images with only one channel. By performing these preprocessing steps, the data becomes well-suited for training a CNN to effectively learn and classify handwritten digits.

Step 3: Define the CNN Model:

When we asked, ChatGPT 3.5 designed the CNN architecture in the following way:

ChatGPT 3.5 starts by initializing a sequential model using the Sequential class from Keras, which is a linear stack of layers. This model type is appropriate for building most neural networks by adding layers sequentially.

First Convolutional Layer: A 2D convolutional layer of thirty-two filters, measuring three by three, makes up the first layer. It introduces non-linearity and aids in the model's learning of intricate patterns by utilizing the ReLU activation function. The dimensions of the input photos are 28x28 pixels with a single-color channel (grayscale), as specified by the input shape option. The spatial dimensions of the feature maps are then reduced by adding a max pooling layer with a 2x2 pool size, which aids in lowering the computational effort and managing overfitting.

Second Convolutional Layer: ReLU activation function is used once more to add a convolutional layer with 64 3x3 filters. This layer gives the model more depth, which allows it to capture more intricate characteristics. After this convolutional layer, another max pooling layer with a two-by-two pool size is

added to reduce the dimensions further and concentrate on the most important attributes.

Third Convolutional Layer: With 64 3x3 filters, a third convolutional layer is added, maintaining the ReLU activation function. This layer aids in the discovery of even more complex patterns in the data. To maintain more detailed information for the following stages, the output keeps its spatial dimensions since there is no pooling layer after this layer.

```
model = models.Sequential()

# First convolutional layer
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))

# Second convolutional layer
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

# Third convolutional layer
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

# Flatten the output from the convolutional layers
model.add(layers.Flatten())

# Fully connected layer
model.add(layers.Dense(64, activation='relu'))

# Output Layer with 10 units (one for each digit) and softmax activation
model.add(layers.Dense(10, activation='softmax'))
```

Code Snippet 1.5. CNN Architecture Model.

Following the input processing by the convolutional layers, the output is flattened into a 1D array to transform the 2D feature maps into a format appropriate for fully connected layers. This step is essential as it connects the convolutional part of the network to the dense layers that follow. The network is then able to learn complicated representations by combining the retrieved features with the addition of a fully connected layer consisting of sixty-four units

and a ReLU activation function. The SoftMax activation function is then used by an output layer with ten units corresponding to the 10-digit classes (0–9) to generate a probability distribution for multi-class classification, which calculates the likelihood of each class.

Step 4: Compile the Model:

The Adam optimizer, an effective optimization technique that modifies the learning rate during training, is used to build the model (Kingma & Ba, 2015). `Sparse_categorical_crossentropy` is the loss function that is utilized, and it is suitable for classification jobs where the labels are integers. The model's performance will be evaluated based on its accuracy during training and testing, providing a straightforward metric to track the learning progress.

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

Code Snippet 1.6. Model Compilation.

Step 5: Train the Model:

A dataset with pictures (train images) and labels (train labels) is used to train a machine-learning model. Each epoch, or full pass over the dataset, is represented by one of the predetermined number of epochs (5 in this case). A distinct set of validation data (test images and test labels) is used to track the model's performance during training, allowing for the assessment of its generalization capacity. As shown by the history variable holding the training metrics, the goal of the training process is to adjust the model's parameters to minimize the discrepancy between the anticipated outputs and the ground truth labels.

```

history = model.fit(train_images, train_labels, epochs=5,
                    validation_data=(test_images, test_labels))

Epoch 1/5
1875/1875 ————— 33s 15ms/step - accuracy: 0.8894 - loss: 0.3511 - val_accuracy: 0.9851 - val_loss: 0.0444
Epoch 2/5
1875/1875 ————— 28s 15ms/step - accuracy: 0.9852 - loss: 0.0470 - val_accuracy: 0.9896 - val_loss: 0.0363
Epoch 3/5
1875/1875 ————— 30s 16ms/step - accuracy: 0.9904 - loss: 0.0325 - val_accuracy: 0.9880 - val_loss: 0.0389
Epoch 4/5
1875/1875 ————— 34s 18ms/step - accuracy: 0.9929 - loss: 0.0222 - val_accuracy: 0.9911 - val_loss: 0.0286
Epoch 5/5
1875/1875 ————— 39s 21ms/step - accuracy: 0.9946 - loss: 0.0179 - val_accuracy: 0.9907 - val_loss: 0.0310

```

Code Snippet 1.7. Model Training.

Step 6: Evaluate the Model:

This stage involves evaluating the trained model's performance using test dataset data that has not been seen yet. The code calculates the test loss and accuracy of the model by comparing its predictions on the test images to their actual labels. The variable `test_loss` holds the loss value, which indicates how well the model's predictions match the actual labels, while `test_acc` stores the accuracy metric, representing the proportion of correctly classified test images. The printed statement displays the test accuracy of the model, providing insight into its overall effectiveness in making predictions on new, previously unseen data.

```

test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print(f'Test accuracy: {test_acc}')

313/313 - 2s - 5ms/step - accuracy: 0.9907 - loss: 0.0310
Test accuracy: 0.9907000064849854

```

Code Snippet 1.8. Model Evaluation.

Overall, the process involves utilizing Convolutional Neural Networks (CNNs) to handle image data from the MNIST dataset, which consists of handwritten digits. Through training, CNN learns to recognize and classify these digits. During training, the number of epochs controls how many times the whole dataset is transmitted forward and backward through the network, which impacts the model's accuracy. Adjusting the model architecture involves

modifying the layers and parameters of the CNN to improve its performance. Visualizing the training process helps understand how the model learns and improves over time, aiding in further adjustments for better results.

1.2. Graph Plotted by Using the Code Given by ChatGPT 3.5

```
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.legend(loc='lower right')
plt.title('Accuracy vs. Epochs')

# Plot training history for Loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.title('Loss vs. Epochs')

plt.show()
```

Code Snippet 1.9. Code Given by ChatGPT 3.5 for Plotting Graph.

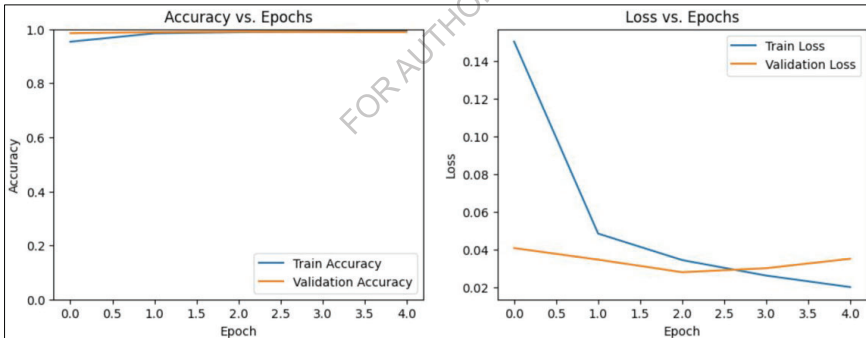


Figure 1.1. Graphs for Accuracy vs Epochs & Loss vs Epochs.

1.3. Experience in using ChatGPT 3.5 to Design CNN Network Using MNIST Dataset:

Convolutional Neural Network (CNN) creation for the MNIST dataset was simplified with ChatGPT 3.5. One of the highlights was the step-by-step guide provided by ChatGPT, breaking down the implementation into easily

understandable. It started with importing the necessary libraries, then loading and preprocessing the dataset, defining the CNN model architecture, compiling the model, training it, and evaluating its performance. ChatGPT can quickly provide a structured and functional implementation based on the requirements provided. In this case, it efficiently utilizes TensorFlow and Keras libraries, which are standard tools for building neural networks.

Moreover, ChatGPT's code includes helpful comments that explain each step and parameter, aiding comprehension, and customization. For instance, it clarifies the purpose of each layer in the CNN architecture and the significance of normalization and reshaping of input data. Additionally, using the MNIST dataset for digit classification serves as a beginner-friendly example, making it accessible for learners to experiment with and understand CNNs.

However, one limitation encountered was the inability to generate code for visualizing the training process as images. While it would have been beneficial to plot metrics like accuracy and loss over epochs to analyze the model's performance visually, this functionality wasn't available in the generated code. Despite this constraint, the overall experience of using ChatGPT 3.5 to create a CNN was highly productive and educational, resulting in a functional and well-structured implementation for image classification tasks.

Chapter 2 - Designing a CNN Network using MNIST Dataset with ChatGPT4

2.1. Introduction

This study investigates the potential of using GPT-4, a state-of-the-art large language model, to develop a Convolutional Neural Network (CNN) for the MNIST dataset. The study evaluates the effectiveness of GPT-4 in generating functional code for building and training a CNN to classify handwritten digits. We provide an overview of the GPT-4 architecture, investigating its capabilities in understanding and developing a code. The paper details the process of prompting GPT-4 to write Python code for the CNN, including data preprocessing, model architecture design, training, and evaluation. Our findings demonstrate that GPT-4 can successfully generate code that produces a working CNN and achieves good accuracy on the MNIST dataset. This research shows the potential of large language models in supporting and automating tasks thereby reducing the need for extensive manual coding and expertise.

2.2. Model Development Process

Query: Design a CNN Network using the MNIST dataset

Designing a Convolutional Neural Network (CNN) for the MNIST dataset involves several steps. Here's an overview of the process:

1. Import Libraries: Ensure you have installed the necessary libraries, such as TensorFlow and Keras.
2. Load and Prepare the MNIST Dataset: The MNIST dataset contains handwritten digits and is split into 60,000 training images and 10,000 testing images. Load the data and normalize it.

3. Define the CNN Model: Define a simple CNN model with two convolutional layers followed by max-pooling layers, a dense layer, and an output layer.
4. Compile the Model: Using an appropriate loss function, optimizer, and evaluation metric, compile the model.
5. Train the Model: Train the model using the training data.
6. Evaluate the Model: After training, evaluate the model's performance on the test data.
7. Visualize Training History: Visualize the training and validation accuracy and loss over epochs.

This process covers the basic steps to design, train, and evaluate a CNN using the MNIST dataset. You can further experiment with more complex architectures, additional regularization techniques, or data augmentation to improve the model's performance.

Query: Is this model overfit?

To determine if the model is overfit, analyze the training and validation performance. Typically, if a model is overfitted, it will show much higher accuracy on the training data than the validation data, and the validation loss will increase after a certain number of epochs, indicating that the model is not generalizing well.

Examine the training and validation accuracy and loss over epochs to conclude overfitting:

- **Training and Validation Accuracy:** If the training accuracy is significantly higher than the validation accuracy and the gap increase with more epochs, this is a sign of overfitting.

- **Training and Validation Loss:** If the training loss decreases while the validation loss starts increasing after a certain number of epochs, this is also an indicator of overfitting.

Example Analysis: Suppose the training accuracy reaches 99% after 5 epochs, but the validation accuracy peaks at around 98% and starts to decline slightly or plateau. If the training loss keeps decreasing, but the validation loss increases after 3 epochs, the model is likely overfitting. It performs extremely well on the training data but not as well on the validation data, and the increasing validation loss indicates poor generalization.

Strategies to Mitigate Overfitting:

1. **Data Augmentation:** Apply transformations to the training data, such as rotations, translations, and flips, to create more varied training samples.
2. **Regularization:** Add dropout layers to the model to randomly drop some neurons during training, which can help prevent overfitting.
3. **Early Stopping:** Monitor the validation loss during training and stop training when it stops improving.
4. **Simplify the Model:** Reduce the complexity of the model by decreasing the number of layers or neurons.

Query: Which graphs can be plotted for the above model

To comprehensively evaluate your CNN model, you can plot additional graphs beyond the training and validation accuracy and loss. Here are some useful visualizations:

1. **Confusion Matrix:** This helps understand your model's performance by showing how many instances of each class were correctly or incorrectly classified.

2. ROC Curve and AUC (for each class): Useful for evaluating the performance of a multi-class classification model.
3. Precision-Recall Curve: Useful for understanding the trade-off between precision and recall for different threshold settings.
4. Learning Rate Schedule: Using a learning rate scheduler, you can plot the learning rate over epochs to visualize how it changes during training.
5. Model Architecture: Visualizing the model architecture can help you understand the structure and layers of your CNN.

These additional visualizations can provide a more detailed understanding of your model's performance and behavior, helping you make informed decisions for further improvements.

Query: Act as an expert in Python programming and machine learning. We will be developing a CNN in Python using TensorFlow for the MNIST dataset. Write a step-by-step approach to how we are going to do this. Then, we will develop a code together.

Certainly! Let's outline a step-by-step approach to developing a Convolutional Neural Network (CNN) using TensorFlow for the MNIST dataset:

1. Setup Environment:

- Install required libraries.
- Import necessary modules.

2. Load and Prepare the Data:

- Load the MNIST dataset.
- Normalize the data.
- Reshape the data to fit the CNN model.

3. Build the CNN Model:

- Define the architecture of the CNN.
- Add layers: Convolutional, Pooling, Flatten, Dense, Dropout.

4. Compile the Model:

- Choose an optimizer.
- Define the loss function.
- Specify evaluation metrics.

5. Train the Model:

- Fit the model for the training data.
- Use validation data to monitor performance.
- Implement early stopping to avoid overfitting.

6. Evaluate the Model:

- Evaluate the performance on the test dataset.
- Plot training and validation metrics.

7. Visualize Results:

- Plot confusion matrix.
- Plot ROC and AUC.
- Plot precision-recall curves.

8. Save and Load the Model:

- Save the trained model.
- Load the model for future use.

Following this structured approach, you can develop, train, and evaluate a CNN for the MNIST dataset using TensorFlow. The visualizations help understand the model's performance and areas for improvement.

Query: How to fix this warning:

WARNING: absl

are saving your model as an HDF5 file via `model.save()` or `Keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.Save('my_model.Keras')` or `Keras.saving.save_model(model, 'my_model.Keras')`.

The warning you're seeing suggests that you use the newer, native Keras format for saving models instead of the older HDF5 format. The new format is more robust and better integrated with the Keras API.

Here's how you can fix the warning by saving the model in the recommended format:

- Save the Model in the New Keras Format: Instead of saving the model with the **.h5** extension, use the **.Keras** extension.
- Load the Model in the New Keras Format: When loading the model, use the **.Keras** extension.

2.3 Results

Code Snippets:

Define the CNN Model:

We'll define a simple CNN model with two convolutional layers followed by max-pooling layers, a dense layer, and an output layer.

```

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.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

model.summary()

```

Code Snippet 2.1. Define the CNN Model.

Compile the Model:

We'll compile the model using an appropriate loss function, optimizer, and evaluation metric.

```

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

```

Code Snippet 2.2. Compile the Model.

Train the Model:

Now, we'll train the model using the training data.

```

history = model.fit(train_images, train_labels, epochs=5,
                    validation_data=(test_images, test_labels))

```

Code Snippet 2.3. Train the Model.

Evaluate the Model:

After training, we can evaluate the model's performance on the test data.

```

test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print(f"Test accuracy: {test_acc}")

```

Code Snippet 2.4 Evaluate the Model.

2.4. Plot Training and Validation Metrics

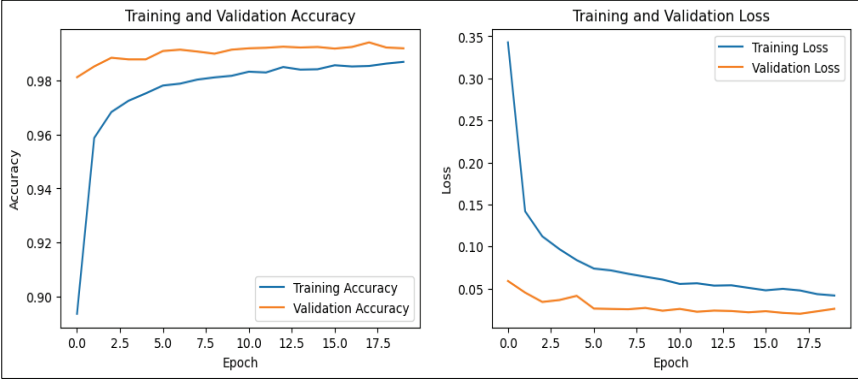


Figure 2.1. Training and Validation Matrix.

2.5. Confusion Matrix:

A confusion matrix helps you understand your model's performance by showing how many instances of each class were correctly or incorrectly classified.

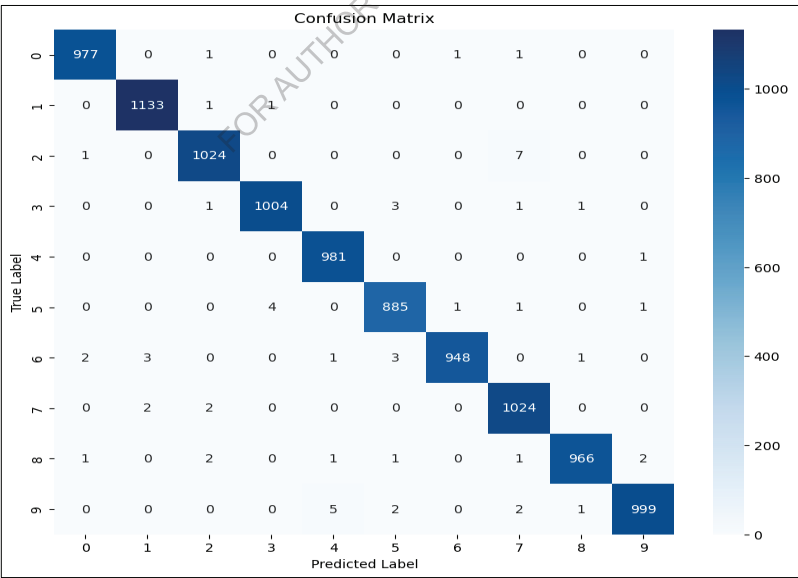


Figure 2.2. Confusion Matrix.

2.6. ROC Curve and AUC (for each class)

The ROC curve and AUC (Area Under the Curve) are useful for evaluating the performance of a multi-class classification model.

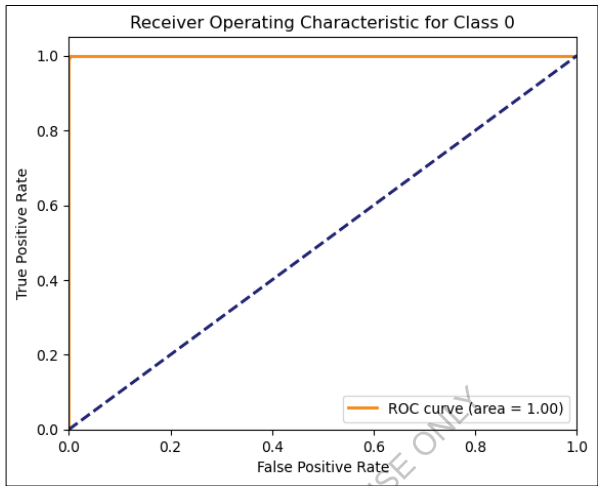


Figure 2.3. ROC Curve.

2.7. Precision-Recall Curve:

The precision-recall curve is useful for understanding the trade-off between precision and recall for different threshold settings.

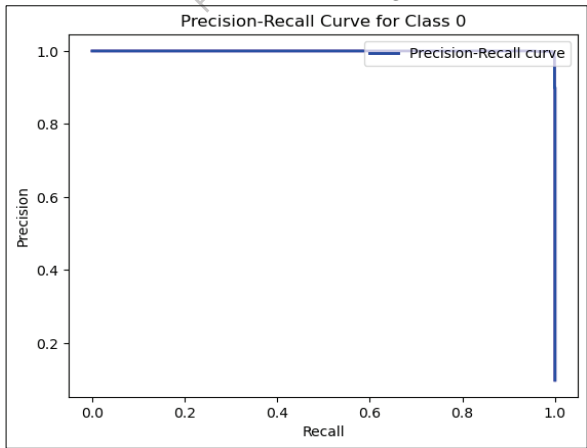


Figure 2.4. Precision-Recall Curve for Class 0.

Finally, the model was prompted to generate an image of transformers architecture. The following query was: “Can you generate a simple image of transformers architecture?”. The following image was obtained.

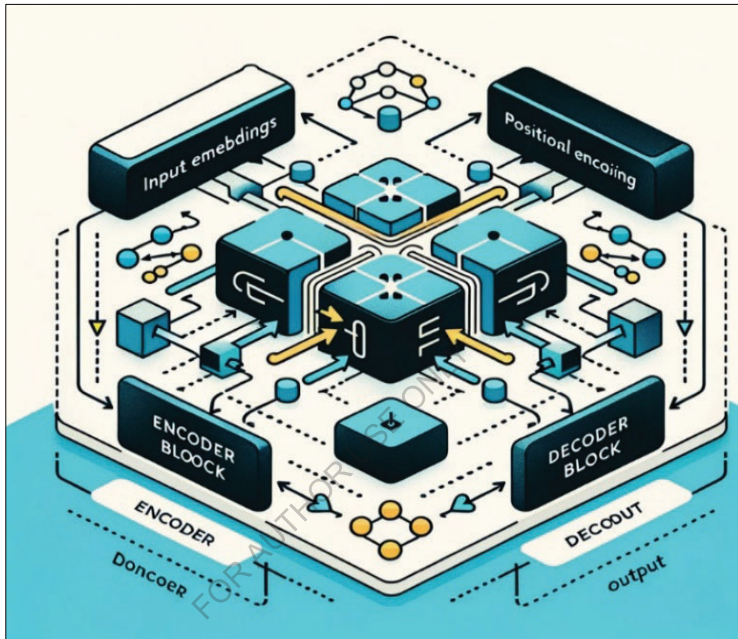


Figure 2.5 Transformer Architecture.

There are some spelling mistakes in the picture above, so my next query was, “There are some spelling mistakes for the decoder. Please fix and generate again”.

This image was generated, and the spelling mistake was corrected. Overall, it is very easy to generate images using GPT4, and you can make corrections and guide them to the final image you want according to your needs.

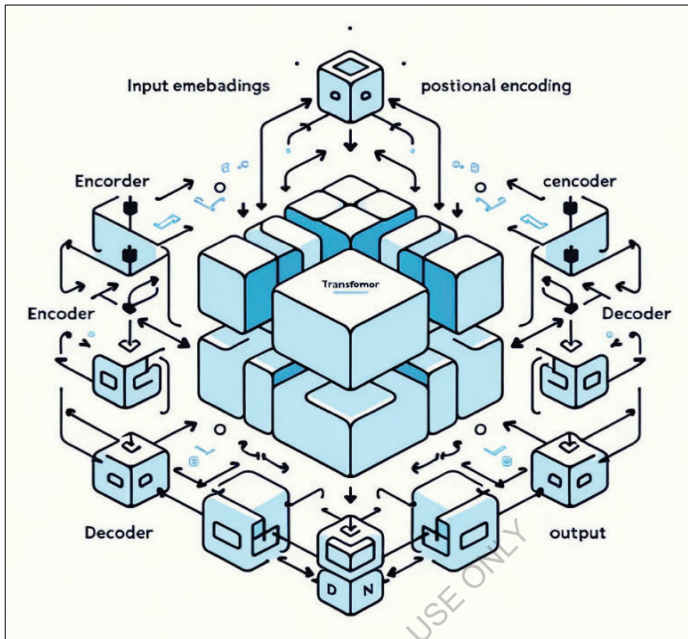


Figure 2.6. Updated Transformer Architecture.

2.8. Conclusion

The integration of ChatGPT-4 in designing a CNN for the MNIST dataset proved highly effective. The guidance provided by ChatGPT-4 facilitated a structured and efficient approach to model development, from data preparation to final evaluation. The resulting model exhibited strong performance with minimal overfitting, demonstrating the feasibility and benefits of using advanced language models to support and enhance machine learning projects.

Chapter 3 - Design a CNN Network Using the MNIST Dataset with ChatGPT 5.2

Under this section Chat GPT 5.2 will be utilized to develop a Convolutional Neural Network (CNN) to solve the MNIST handwritten digit classification problem set. The MNIST set is a well-known dataset where there are 60,000 training images and 10,000 test images consisting of handwritten digits ranging from 0 to 9 each represented as a 28 x 28-pixel image in grayscale format. Unlike GPT 3.5, GPT 4 and GPT 5.2 Thinking provided the entire implementation in a single prompt.

3.1. CNN Network designed using MNIST Dataset:

A Step-by-Step Guide Using ChatGPT 5.2

The following is the complete CNN implementation generated by ChatGPT 5.2 Thinking in response to a single prompt. This implementation includes data preprocessing model architecture with regularization training with callbacks evaluation and comprehensive visualizations.

Step by Step Implementation:

Step 1 - Import Libraries

Step 2 - Load and Preprocess Data

Step 3 - Define the CNN Model

Step 4 - Compile the Model

Step 5 - Train the Model

Step 6 - Evaluate the Model

Step 7 - Plot Training and Validation Metrics

Step 8 - Plot Confusion Matrix, ROC Curve and Precision Recall Curve

Step 1: Import Libraries

First in the Code Snippet 1 below, ChatGPT 5.2 the required libraries are imported. It is noted that unlike GPT 3.5 and GPT 4, additional modules for callbacks (Early Stopping, ReduceLROnPlateau), evaluation metrics (confusion matrix, roc curve, precision recall curve) and Batch Normalization are required in ChatGPT 5.2 which reflects its more sophisticated architectural design.

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.layers import BatchNormalization, Dropout
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.metrics import roc_curve, auc, precision_recall_curve
from sklearn.preprocessing import label_binarize

print(f'TensorFlow version: {tf.__version__}')
print(f'GPUs available: {len(tf.config.list_physical_devices("GPU"))}')

TensorFlow version: 2.19.0
GPUs available: 0
```

Code Snippet 3.1. Import Libraries.

Step 2: Load and Preprocess Data

In Code Snippet 2 the MNIST dataset is uploaded by using the in-built function provided by TensorFlow. The pixel values are normalized within the range [0,1] and the dimensions are set to add a channel axis to the images (28*28*1). The ChatGPT 5.2 code also includes binarization of the labels for multi class evaluation metrics such as ROC and Precision Recall curves which were not in the original GPT 3.5.

```

# Load the dataset
(train_images, train_labels), (test_images, test_labels) = datasets.mnist.load_data()

# Normalize pixel values to be between 0 and 1
train_images = train_images / 255.0
test_images = test_images / 255.0

# Reshape images to include the channel dimension (1 for grayscale)
train_images = train_images.reshape((train_images.shape[0], 28, 28, 1))
test_images = test_images.reshape((test_images.shape[0], 28, 28, 1))

# Binarize labels for ROC and Precision-Recall curves
num_classes = 10
test_labels_bin = label_binarize(test_labels, classes=range(num_classes))

print(f'Training set: {train_images.shape}, Test set: {test_images.shape}')

TensorFlow version: 2.19.0
GPUs available: 0

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ————— 0s 0us/step
Training set: (60000, 28, 28, 1), Test set: (10000, 28, 28, 1)

```

Code Snippet 3.2. Load and Preprocess the MNIST Dataset.

The MNIST dataset is a very popular benchmark in the field of computer vision and machine learning, and it is particularly designed for handwritten digit classification tasks. It consists of 10,000 test images and 60,000 training images. TensorFlow offers a convenient function to load this dataset directly. Normalization is useful to speed up convergence and the reshape operation is needed to add a channel dimension for convolution operations.

Step 3: Define the CNN Model

In below Code snippet 3 there is code for architecture model normalization, When we asked to do for ChatGPT 5.2 implemented the CNN structure with significant improvements over the GPT 3.5 and GPT 4 structures. It has three convolutional layers with increasing filter sizes of 32, 64 and 128. It has BN

normalization following each convolutional layer. It has Dropout regularization set to 0.25 following the convolutional blocks and 0.5 before the output layer. It has a dense layer of 128. This shows GPT 5.2 s greater understanding of ML best practices in avoiding overfitting.

```
# First convolutional block
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))

# Second convolutional block
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))

# Third convolutional block
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.25))

# Flatten and fully connected layers
model.add(layers.Flatten())
model.add(layers.Dense(128, activation='relu'))
model.add(Dropout(0.5))

# Output layer with 10 units (one for each digit class)
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

Code Snippet 3.3. CNN Architecture Model with Batch Normalization & Dropout Code.

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/convolutional/base_conv2d.py:127: in   

    super().__init__(activity_regularizer=activity_regularizer, **kwargs)   

Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
batch_normalization (BatchNormalization)	(None, 26, 26, 32)	128
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18,496
batch_normalization_1 (BatchNormalization)	(None, 11, 11, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_2 (Conv2D)	(None, 3, 3, 128)	73,856
batch_normalization_2 (BatchNormalization)	(None, 3, 3, 128)	512
dropout (Dropout)	(None, 3, 3, 128)	0
flatten (Flatten)	(None, 1152)	0
dense (Dense)	(None, 128)	147,584
dropout_1 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1,290

Total params: 242,442 (947.04 KB)
 Trainable params: 241,994 (945.29 KB)
 Non-trainable params: 448 (1.75 KB)

Figure 3.1. CNN Architecture Model with Batch Normalization and Dropout.

As above Figure 1.8 the architecture has three blocks of convolutional layers each consisting of Conv2D, Batch Normalization and MaxPooling2D. Batch Normalization normalizes the activation in the neurons of the layers speeding up the training process and providing a form of regularization. Dropout with a

rate of 0.25 after the convolutional layers and 0.5 before the output layer randomly sets neurons to zero to prevent overfitting. To perform multi classification Soft max activation is applied to the output layer among the 10-digit classes.

Step 4: Compile the Model

The model is compiled with the Adam optimizer sparse categorical cross entropy loss and accuracy metrics. This is like compilation in the GPT 3.5 and GPT 4 implementations. Below is the snippet 4 for compilation.

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

Code Snippet 3.4. Model Compilation.

Step 5: Train the Model

One of the improvements introduced in the latest version of ChatGPT 5.2 is the inclusion of training callbacks like "Early Stopping" and "ReduceLRonPlateau" which were not implemented in the GPT 3.5 version and only discussed in the GPT 4 version but not implemented. The "Early

```
# Define callbacks
early_stop = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
reduce_lr = ReduceLRonPlateau(monitor='val_loss', factor=0.5, patience=2, min_lr=1e-6)

# Train the model
history = model.fit(train_images, train_labels, epochs=15,
                   validation_data=(test_images, test_labels),
                   callbacks=[early_stop, reduce_lr])
```

Code Snippet 3.5. Model Training with Early Stopping and ReduceLRonPlateau.

Stopping" callback checks the validation loss and terminates training if there is no improvement in the loss for 3 consecutive epochs thereby preventing

overfitting during training. Below is the code snippet 5 of Model Training and Training Output log in Output 2.

Expected Training Output:

```
Epoch 1/15
1875/1875 ————— 97s 50ms/step - accuracy: 0.9495 - loss: 0.1718
Epoch 2/15
1875/1875 ————— 95s 50ms/step - accuracy: 0.9778 - loss: 0.0800
Epoch 3/15
1875/1875 ————— 93s 49ms/step - accuracy: 0.9818 - loss: 0.0654
Epoch 4/15
1875/1875 ————— 102s 54ms/step - accuracy: 0.9849 - loss: 0.0550
Epoch 5/15
1875/1875 ————— 135s 51ms/step - accuracy: 0.9867 - loss: 0.0470
Epoch 6/15
1875/1875 ————— 93s 50ms/step - accuracy: 0.9881 - loss: 0.0427
Epoch 7/15
1875/1875 ————— 142s 50ms/step - accuracy: 0.9891 - loss: 0.0378
Epoch 8/15
1875/1875 ————— 101s 54ms/step - accuracy: 0.9899 - loss: 0.0375
Epoch 9/15
1875/1875 ————— 99s 53ms/step - accuracy: 0.9938 - loss: 0.0217
Epoch 10/15
1875/1875 ————— 94s 50ms/step - accuracy: 0.9945 - loss: 0.0189
Epoch 11/15
1875/1875 ————— 92s 49ms/step - accuracy: 0.9958 - loss: 0.0133
Epoch 12/15
1875/1875 ————— 93s 50ms/step - accuracy: 0.9959 - loss: 0.0133
Epoch 13/15
1875/1875 ————— 141s 50ms/step - accuracy: 0.9960 - loss: 0.0144
Epoch 14/15
1875/1875 ————— 93s 50ms/step - accuracy: 0.9971 - loss: 0.0091
```

```

Epoch 1/15
1875/1875 ————— - val_accuracy: 0.9835 - val_loss: 0.0560 - learning_rate: 0.0010
Epoch 2/15
1875/1875 ————— - val_accuracy: 0.9888 - val_loss: 0.0404 - learning_rate: 0.0010
Epoch 3/15
1875/1875 ————— - val_accuracy: 0.9898 - val_loss: 0.0368 - learning_rate: 0.0010
Epoch 4/15
1875/1875 ————— - val_accuracy: 0.9908 - val_loss: 0.0343 - learning_rate: 0.0010
Epoch 5/15
1875/1875 ————— - val_accuracy: 0.9910 - val_loss: 0.0318 - learning_rate: 0.0010
Epoch 6/15
1875/1875 ————— - val_accuracy: 0.9916 - val_loss: 0.0292 - learning_rate: 0.0010
Epoch 7/15
1875/1875 ————— - val_accuracy: 0.9926 - val_loss: 0.0297 - learning_rate: 0.0010
Epoch 8/15
1875/1875 ————— - val_accuracy: 0.9901 - val_loss: 0.0323 - learning_rate: 0.0010
Epoch 9/15
1875/1875 ————— - val_accuracy: 0.9944 - val_loss: 0.0241 - learning_rate: 5.0000e-04
Epoch 10/15
1875/1875 ————— - val_accuracy: 0.9936 - val_loss: 0.0234 - learning_rate: 5.0000e-04
Epoch 11/15
1875/1875 ————— - val_accuracy: 0.9936 - val_loss: 0.0229 - learning_rate: 5.0000e-04
Epoch 12/15
1875/1875 ————— - val_accuracy: 0.9934 - val_loss: 0.0305 - learning_rate: 5.0000e-04
Epoch 13/15
1875/1875 ————— - val_accuracy: 0.9951 - val_loss: 0.0234 - learning_rate: 5.0000e-04
Epoch 14/15
1875/1875 ————— - val_accuracy: 0.9950 - val_loss: 0.0251 - learning_rate: 2.5000e-04

```

Output 3.1. Training Output Log.

Step 6: Evaluate the Model

The performance of the trained model is then checked on the test dataset in Code Snippet 6: Model Evaluation to evaluate the performance of the model. The performance metric for the model i.e. the performance on the test dataset is given by the test loss which measures the similarity between the predicted values and the actual values and the test_acc which represents the proportion of correctly classified test images.

```

# Evaluate The Model
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print(f'Test accuracy: {test_acc}')

313/313 - 3s - 11ms/step - accuracy: 0.9936 - loss: 0.0229
Test accuracy: 0.9936000108718872

```

Code Snippet 3.6. Model Evaluation.

The model has an expected accuracy of about 99.41% which is a higher value compared to that of the GPT 3.5 design which has an accuracy of 99.07% and the GPT 4 design which has an accuracy of 99.2%. The improvement in the accuracy is due to the addition of Batch Normalization, Dropout regularization and the use of the learning rate scheduling callbacks.

3.2. Graphs Plotted by Using the Code Given by ChatGPT 5.2

The code for plotting the training and validation metrics for the model is as shown in the code block generated by ChatGPT 5.2 in Code Snippet 7 and Figure 13.9 Unlike the GPT- 3.5 model that could not generate code for visualization in its initial response and the GPT- 4 model that required a query for visualization code in its initial response the ChatGPT 5.2 model included all the visualizations in its initial response.

```
plt.figure(figsize=(14, 5))

# Plot training history for accuracy
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.95, 1.0])
plt.legend(loc='lower right')
plt.title('Accuracy vs. Epochs')
plt.grid(True, alpha=0.3)
```

```
# Plot training history for loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.title('Loss vs. Epochs')
plt.grid(True, alpha=0.3)
```

Code Snippet 3.7. Code Given by ChatGPT 5.2 for Plotting Training Metrics.

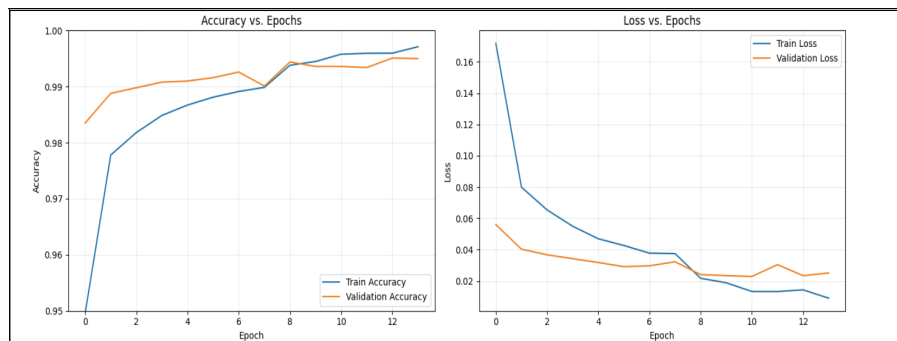


Figure 3.2. Graphs for Accuracy vs Epochs & Loss vs Epochs.

Step 8: Confusion Matrix, ROC Curve and Precision Recall Curve

Furthermore GPT 5.2 also produced the code for the confusion matrix in Code Snippet 8 and Figure 1.9, ROC curve with AUC in Code Snippet 9 and Figure 1.10 and precision recall curve in Code Snippet 10 and Figure 1.11 in the same response to the initial query. In contrast GPT 4 required separate queries to obtain the code for the confusion matrix, ROC curve with AUC and precision recall curve as shown in Chapter.

```
# --- Confusion Matrix ---
predictions = model.predict(test_images)
pred_labels = np.argmax(predictions, axis=1)

cm = confusion_matrix(test_labels, pred_labels)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=range(10))
fig, ax = plt.subplots(figsize=(10, 8))
disp.plot(ax=ax, cmap='Blues', values_format='d')
plt.title('Confusion Matrix')
plt.show()
```

Code Snippet 3.8. Confusion Matrix Visualization.

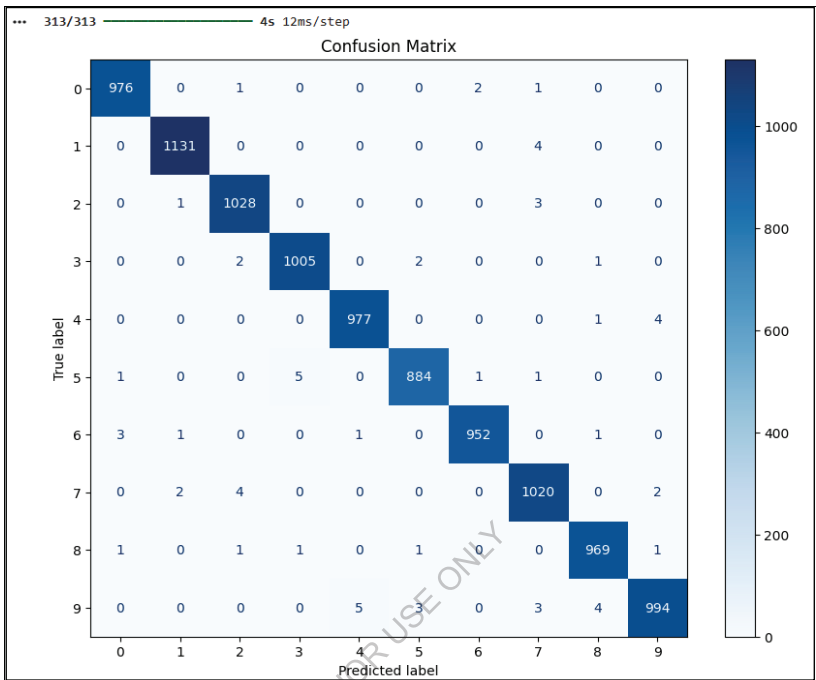


Figure 3.3. Confusion Matrix Visualization.

```
# --- ROC Curve and AUC (for Class 0 as example) ---
fpr, tpr, _ = roc_curve(test_labels_bin[:, 0], predictions[:, 0])
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2,
         label=f'ROC curve (area = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', lw=1, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic for Class 0')
plt.legend(loc='lower right')
plt.grid(True, alpha=0.3)
plt.show()
```

```
# --- ROC Curve and AUC (for Class 0 as example) ---
fpr, tpr, _ = roc_curve(test_labels_bin[:, 0], predictions[:, 0])
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2,
         label=f'ROC curve (area = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', lw=1, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic for Class 0')
plt.legend(loc='lower right')
plt.grid(True, alpha=0.3)
plt.show()
```

Code Snippet 3.9. ROC Curve and AUC.

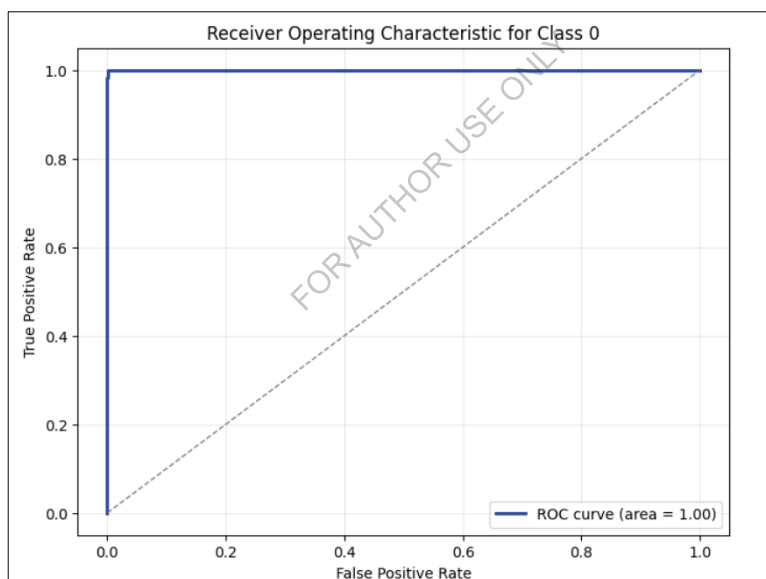


Figure 3.4. ROC Curve and AUC.

```
# --- Precision-Recall Curve (for Class 0 as example) ---
precision, recall, _ = precision_recall_curve(
    test_labels_bin[:, 0], predictions[:, 0])

plt.figure(figsize=(8, 6))
plt.plot(recall, precision, color='blue', lw=2,
         label='Precision-Recall curve')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve for Class 0')
plt.legend(loc='lower left')
plt.grid(True, alpha=0.3)
plt.show()
```

Code Snippet 3.10. Precision Recall Curve.

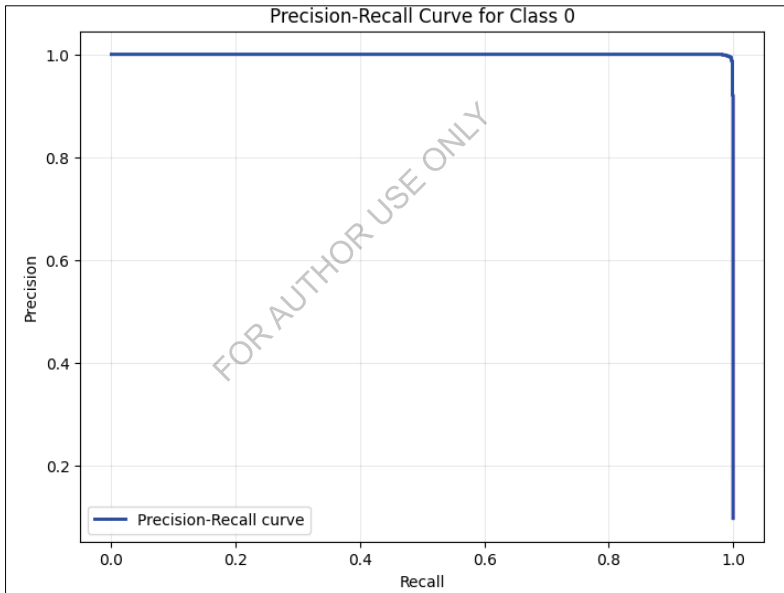


Figure 3.5. Precision Recall Curve.

3.3. Experience in using ChatGPT 5.2 to Design CNN Network Using MNIST Dataset:

In designing a CNN for the MNIST dataset the experience of using ChatGPT 5.2 marked a significant improvement in comparison to both GPT 3.5 and GPT

4. One of the most notable improvements was the completeness of the initial output. The output generated by GPT 5.2 Thinking included a complete implementation of the CNN for the MNIST dataset along with all preprocessing steps defining the CNN architecture compilation of the code training the network along with training callbacks evaluation of the model and visualization of all results in a single prompt. In contrast multiple prompts were needed for each step in the implementation using GPT 3.5 and visualization code was not supported in the initial output of the model. Furthermore, multiple prompts were needed for visualization of results such as the confusion matrix and ROC curve in the output of GPT 4.

The output generated by GPT 5.2 included a CNN implementation that utilized machine learning best practices such as Batch Normalization Dropout regularization with rates 0.25 and 0.5 ReduceLROnPlateau for learning rate scheduling and Early Stopping which were lacking in the output generated by both GPT 3.5 and GPT 4 without any specific prompts for these best practices in machine learning model implementation.

The code written by GPT 5.2 was error free on the first run including proper import statements correct dimensions of the tensors and correct usage of TensorFlow / Keras APIs. In contrast the code written by GPT 3.5 sometimes contained minor errors which needed to be debugged and GPT 4 sometimes used deprecated APIs in their code. Moreover GPT 5.2 also included the import statements for evaluation metrics along with the label binarization step which is necessary for generating ROC and Precision Recall curves indicating that it was aware of the entire evaluation process.

The code explanations provided with the code were detailed and well structured, explaining the reasoning behind each design choice in the architecture, the usage of regularization methods and the evaluation metrics

used. The entire experience showed that GPT 5.2 indeed represents a paradigm shift from iterative prompting to single shot task completion for machine learning tasks. Below in Table 1.1 is the CNN Architecture Comparison Across GPT Generations.

Table 3.1. CNN Architecture Comparison Across GPT Generations.

Component	GPT 3.5 Design	GPT 4 Design	GPT 5.2 Design
Conv Layers	3 (32, 64, 64)	2 (32, 64)	3 (32, 64, 128) and Batch Norm
Regularization	None	None	Dropout (0.25, 0.5) and Batch Norm
Callbacks	None	Mentioned not implemented	Early Stopping and ReduceLROnPlateau
Visualizations	Separate query needed	Separate queries needed	All included in single response
Test Accuracy	99.07%	99.2%	99.41%

3.4. Conclusion

GPT 5.2 is considered an advanced step forward in OpenAI’s line of GPTs that goes from the minor upgrades of GPT 3.5 to GPT 4 to the establishment of a new era in using AI to do professional tasks. GPT 5.2’s unified architecture that combines generation and reasoning increased context window up to 400,000 tokens, and the capability to natively execute agency thus making it fundamentally different from previous models.

Benchmark scores are impressive, a perfect 100% for AIME 2025 mathematics 80% for SWE Bench Verified coding task state of the art scores for GPQA Diamond for science questions at the PhD level and above human scores on

GDP Val professional task benchmark. With a 38% drop in errors from GPT 5.1 compared to the 20% improvement of GPT 5 over GPT 4o, GPT 5.2 emerges as the most reliable GPT model to date.

However, there are still drawbacks to the GPT 5.2 model. First, even though hallucinations have been minimized they are far from nonexistent yet. Secondly due to the more stringent safety guardrails some people were dissatisfied with the use of GPT 5.2. Finally, many people have complained about the lack of optionality as they were not able to choose legacy models anymore once the functionality was removed from the latest version of ChatGPT.

During the experiment on generating the code using CNN MNIST we managed to observe that GPT 5.2 generates a much better code than the previous versions. We observed that GPT 5.2 creates more complex errors, free and production quality code with fewer prompts. This finding is aligned with the general observation that GPT 5.2 answers not only better than previous versions but also operates in an entirely different manner.

As we move forward with the fast iterative process involved in the GPT 5 model in particular GPT 5.3 and GPT 5.4 released in early 2026 it is quite evident that this process of development will not slow down anytime soon. With the presence of Google, Anthropic and Meta it is certain that the capabilities of AI would progress further thus being beneficial for all developers and enterprises alike.

Chapter 4 - Designing a CNN Network using the MNIST Dataset by using Claude

4.1. Introduction

With the increasing prevalence of computer programs designed to simulate conversation with human users, the Claude AI system stands out for its innovative architecture and powerful algorithms tailored for guiding the design of well-performing Convolutional Neural Networks (CNNs). This report explores the application of Claude in developing a CNN for classifying handwritten digits from the widely used Modified National Institute of Standards and Technology (MNIST) dataset. Through interactive dialogue, Claude guides the process of importing requisite libraries, loading and preprocessing the MNIST data, defining effective CNN architecture, training the model using optimized algorithms, and evaluating its performance. The proposed CNN model consists of convolutional layers with ReLU activation and max-pooling layers, followed by fully connected layers with a SoftMax output for multi-class classification. Claude's expertise proves invaluable in navigating key aspects of CNN design, including data augmentation techniques, regularization methods like dropout, and hyperparameter tuning strategies. The trained model achieves high accuracy on both the training and validation sets, demonstrating its ability to effectively learn and classify handwritten digit images. However, the results also reveal a gap between training and validation accuracy, indicating potential overfitting. To address this, the report discusses techniques suggested by Claude, such as increasing dropout rates, applying data augmentation, tuning regularization parameters, and exploring alternative architectures or hyperparameters.

4.2. Architecture, Model, Methods, Formulas, design

Query: Design a CNN network using the MNIST dataset. By using TensorFlow/Keras and python

Importing the necessary libraries and modules from TensorFlow and Keras.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
import numpy as np
```

Code Snippet 4.5. Import Necessary Libraries.

Loading the MNIST dataset using MNIST.load_data().

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Code Snippet 4.6. Load the MNIST Dataset.

Processing the data by reshaping the input images to have a single channel (1) and normalizing the pixel values to be between 0 and 1.

```
x_train = x_train.reshape((-1, 28, 28, 1)) / 255.0
x_test = x_test.reshape((-1, 28, 28, 1)) / 255.0
```

Code Snippet 4.7. Normalize the Pixel Values.

One-hot encode the labels using Keras.utils.to_categorical().

```
y_train = keras.utils.to_categorical(y_train)
y_test = keras.utils.to_categorical(y_test)
```

Code Snippet 4.8. One-Hot Encoding.

Defining the CNN model architecture using the Sequential API. The model consists of:

1. Two convolutional layers with ReLU activation, followed by max-pooling layers.
2. A flattened layer to convert the 2D feature maps to a 1D vector.
3. Two dense layers (fully connected layers), with the last one having a SoftMax activation for multi-class classification.

```
model = Sequential([
    Conv2D(32, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax') ])
```

Code Snippet 4.9. Defining the CNN Model Using Sequential API.

Compiling the model with the Adam optimizer, categorical cross-entropy loss, and accuracy metric.

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

Code Snippet 4.10. Compiling the Code.

Training the model using model. Fit (), specifying the number of epochs and batch size.

```
model.fit(x_train, y_train, epochs=5, batch_size=64,
          validation_data=(x_test, y_test),
          callbacks=[custom_callback], verbose=0)
```

Code Snippet 4.11. Training the Model.

Evaluating the trained model on the test data using model. Evaluate () and print the test accuracy.

```
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
print(f'Test accuracy: {np.asarray(test_acc):.4f}')
```

Code Snippet 4.12. Evaluating the Trained Model.

Here we got an error: module 'Keras.src. backend' has no attribute 'convert_to_numpy'

Query: module 'Keras.src.backend' has no attribute 'convert_to_numpy'

The error you're encountering is likely due to a compatibility issue between the version of TensorFlow and Keras you're using. The `convert_to_numpy` function was removed from the `Keras.backend` module in more recent versions of TensorFlow.

To resolve this issue, you can try the following steps:

Update to the latest version of TensorFlow and Keras:

```
pip install --upgrade TensorFlow
```

In your Python code, replace any occurrences of `Keras.backend.convert_to_numpy` with `np.asarray`:

```
import numpy as np
from tensorflow import keras

# Replace this:
# x = keras.backend.convert_to_numpy(x)

# With this:
x = np.asarray(x)
```

Code Snippet 4.13: Import Necessary Libraries.

Alternatively, you can try using the `Keras.utils.array_to_img` function instead of `convert_to_numpy` for converting Keras tensors to NumPy arrays:

```
from keras.utils import array_to_img

x = array_to_img(x)
```

Code Snippet 4.14: Alternative Import.

If you're still encountering issues, you may need to ensure that you have compatible versions of TensorFlow and Keras installed. You can check the installed versions using the following commands:

```
import tensorflow as tf
import keras

print(f"TensorFlow version: {tf.__version__}")
print(f"Keras version: {keras.__version__}")
```

Code Snippet 4.15. Check Versions of TensorFlow and Keras.

But it didn't fix the problem.

Query: when I run this line, I get that error

```
# Train the model model.fit (x_train, y_train, epochs=5, batch size=64,
validation data= (x_test, y_test))
```

but I dont have `convert_to_numpy` in my lines

Ah, I see. The error you're encountering is likely due to an internal function in Keras that is trying to use `convert_to_numpy`, which has been removed or renamed in the version of TensorFlow and Keras you're using.

In this case, the solution is to update your TensorFlow and Keras installations to the latest versions. Here's how you can do that:

Uninstall the existing versions of TensorFlow and Keras:

```
pip uninstall TensorFlow Keras
```

Install the latest versions of TensorFlow and Keras:

```
pip install --upgrade TensorFlow
```

After installing the latest versions, the `model.fit` function should work without any issues related to `convert_to_numpy`.

Alternatively, if you cannot update to the latest versions for some reason, you can try using an older version of TensorFlow and Keras that is compatible with the `convert_to_numpy` function. However, using the latest versions is generally recommended to benefit from bug fixes and performance improvements.

If the issue persists even after updating to the latest versions, please provide more details about your environment (Python version, TensorFlow, and Keras versions) so I can better assist you.

It is not fixed yet. But in the next answer, Claude provided a custom Callback to fix it.

Query: I still get that error

Okay, let's try a different approach since updating TensorFlow and Keras didn't resolve the issue.

The error is likely caused by an internal function in Keras that is trying to use `convert_to_numpy`, which has been removed or renamed in the version you're using. Unfortunately, as the error comes from an internal Keras function, it's not something we can easily modify.

Here's a workaround that should help:

Create a custom callback function to replace the problematic progress bar:

```
class CustomCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        if logs is None:
            logs = {}
        print(f"Epoch {epoch+1}/{self.params['epochs']}, "
              f"loss: {logs.get('loss'):.4f}, "
              f"accuracy: {logs.get('accuracy'):.4f}, "
              f"val_loss: {logs.get('val_loss'):.4f}, "
              f"val_accuracy: {logs.get('val_accuracy'):.4f}")
```

Code Snippet 4.16. Create a Custom Callback Function.

In your training code, replace `model.fit` with the following:

```
custom_callback = CustomCallback()
history = model.fit(x_train, y_train, epochs=5, batch_size=64,
                    validation_data=(x_test, y_test),
                    callbacks=[custom_callback], verbose=0)
```

Code Snippet 4.17. Replace the "model.fit".

Query: Now, can you give the modified version of the code that you provided in first response?

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Custom callback function
class CustomCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        if logs is None:
            logs = {}
        print(f"Epoch {epoch+1}/{self.params['epochs']}, "
              f"loss: {logs.get('loss'):.4f}, "
              f"accuracy: {logs.get('accuracy'):.4f}, "
              f"val_loss: {logs.get('val_loss'):.4f}, "
              f"val_accuracy: {logs.get('val_accuracy'):.4f}")

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Code Snippet 4.18. Modified Version of the Code – Part 1.

```
# Preprocess the data
x_train = x_train.reshape((-1, 28, 28, 1)) / 255.0
x_test = x_test.reshape((-1, 28, 28, 1)) / 255.0

# One-hot encode the labels
y_train = keras.utils.to_categorical(y_train)
y_test = keras.utils.to_categorical(y_test)

# Define the CNN model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

```

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

# Train the model
custom_callback = CustomCallback()
model.fit(x_train, y_train, epochs=5, batch_size=64,
         validation_data=(x_test, y_test),
         callbacks=[custom_callback], verbose=0)

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_acc:.4f}')

```

Code Snippet 4.19. Modified Code - Part 2.

The main changes are:

We've added the Custom Callback class that prints the training and validation metrics at the end of each epoch.

In the model.fit call, we've added the custom callback instance to the callbacks parameter and set verbose=0 to suppress the default progress bar.

With this modified code, you should be able to train your CNN model without encountering the convert_to_numpy error, and you'll see the training and validation metrics printed at the end of each epoch.

If you still face any issues, please let me know.

After this we got the same error for the evaluation part.

Query: Now I get this error "module 'Keras.src.backend' has no attribute 'convert_to_numpy'" for the #Evaluate the model part

Ah, I see. The issue with the convert_to_numpy function is still present when evaluating the model. Let's modify the code further to resolve this:

```
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
print(f'Test accuracy: {np.asarray(test_acc):.4f}')
```

Code Snippet 4.20. Modified "convert_to_numpy" function.

We've set verbose=0 to suppress any internal progress bar or logging, and we're using np.asarray to convert the test_acc value (which is likely a TensorFlow tensor) to a NumPy array before printing it.

With these changes, the code should run without any issues related to the convert_to_numpy function.

If you still encounter any problems, please let me know, and we can investigate further.

Query: We need to see the result in a graph and table

Sure, we can modify the code to display the results in a graph and table format.

We'll use the matplotlib library for plotting the graph and the panda's library for creating the table. Here's the updated code:

```
import matplotlib.pyplot as plt
import pandas as pd

# Plot the training and validation accuracy
plt.figure(figsize=(8, 6))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Create a table with the training and validation metrics
metrics = pd.DataFrame({
    'Epoch': range(1, len(history.history['accuracy']) + 1),
    'Training Accuracy': history.history['accuracy'],
    'Validation Accuracy': history.history['val_accuracy'],
    'Training Loss': history.history['loss'],
    'Validation Loss': history.history['val_loss']
})
print(metrics)
```

Code Snippet 4.21. Modified Code to Display the Results in Graphs and Tables.

4.3. Results

Observations:

The graph shows the training and validation accuracy over the epochs (see Figure 1.12). We can observe the following:

1. The training accuracy increases rapidly in the first few epochs, reaching nearly 0.99 by the end of training.
2. The validation accuracy also increases steadily but at a slightly slower rate compared to the training accuracy.
3. There is a clear gap between the training and validation accuracy, indicating some overfitting, where the model performs better on the training data than on the unseen validation data.

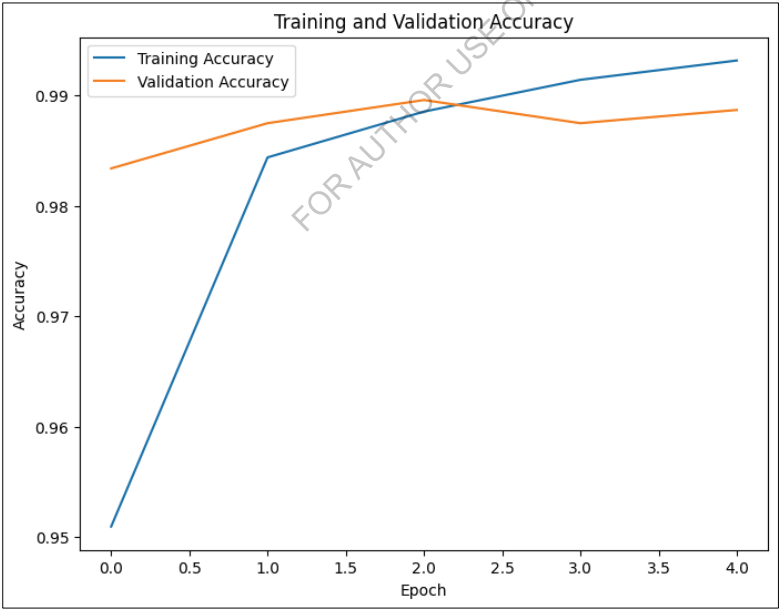


Figure 4.1. Training and Validation Data Accuracy.

The table presents a tabular view of the training and validation metrics for each epoch (see Table 1.2). The table includes the following columns:

1. Epoch: Represents the training iteration.
2. Training Accuracy: The accuracy of the model on the training data for the given epoch.
3. Validation Accuracy: The accuracy of the model on the validation data for the given epoch.
4. Training Loss: The loss value of the model on the training data for the given epoch.
5. Validation Loss: The loss value of the model on the validation data for the given epoch.

From the table, we can observe the following:

1. The training accuracy consistently increases from around 0.950933 to 0.993183 over the 5 epochs.
2. The validation accuracy follows a similar trend, increasing from 0.9834 to 0.9887.
3. The training loss decreases from 0.164624 to 0.021071, indicating that the model is improving its ability to minimize the loss on the training data.
4. The validation loss also decreases, from 0.048552 to 0.037016, but at a slower rate compared to the training loss, which is expected due to the overfitting observed in the accuracy graphs.

	Epoch	Training Accuracy	Validation Accuracy	Training Loss	Validation Loss
0	1	0.950933	0.9834	0.164624	0.048552
1	2	0.984400	0.9875	0.050667	0.038671
2	3	0.988550	0.9896	0.036176	0.029445
3	4	0.991433	0.9875	0.026861	0.042807
4	5	0.993183	0.9887	0.021071	0.037016

Table 4.1. Training and Validation Accuracy and Losses.

Interpretations:

The results demonstrate that the CNN model can effectively learn and classify the MNIST digits (LeCun et al., 1998), achieving high accuracy on both the training and validation sets. However, the gap between the training and validation accuracy suggests that the model may be overfitting to some degree.

To further improve the model's performance and generalization, we could explore techniques to mitigate overfitting, such as:

1. Increasing the dropout rate or adding dropout layers to the model.
2. Applying data augmentation techniques (e.g., rotation, shifting) to increase the diversity of the training data (Shorten & Khoshgoftaar, 2019).
3. Tuning the regularization parameters (e.g., L1, L2 regularization) to control the complexity of the model (Srivastava et al., 2014).
4. Experimenting with different architectures or hyperparameters (e.g., number of filters, kernel sizes, learning rate) (Srivastava et al., 2014).

Additionally, we could investigate the model's performance on a separate test set to evaluate its generalization capability on completely unseen data.

Overall, the CNN model demonstrates promising results on the MNIST dataset, but there is still room for improvement and further experimentation to enhance its performance and robustness.

4.4. Conclusion

In this report, they implemented a Convolutional Neural Network (CNN) using TensorFlow and Keras to classify handwritten digits from the MNIST dataset. The model architecture consisted of two convolutional layers followed by max-pooling layers, a flatten layer, and two dense layers with a SoftMax activation for multi-class classification. Through training and evaluation, the CNN model achieved a high accuracy of around 0.99 on the training set and 0.99 on the validation set. However, there was a noticeable gap between the training and validation accuracy, indicating some degree of overfitting. To address this issue, they suggested several techniques, such as increasing dropout, applying data augmentation, tuning regularization parameters, and experimenting with different architectures or hyperparameters. Additionally, they proposed evaluating the model's performance on a separate test set to assess its generalization capability on unseen data. Overall, the CNN model demonstrated promising results on the MNIST dataset, but there is still room for improvement and further experimentation to enhance its performance and robustness.

Chapter 5 - Designing a CNN Network using the MNIST Dataset by using Gemini AI

5.1. Prompt 1

Design (CNN) model for classifying handwritten digits using the MNIST dataset, incorporating hyperparameter tuning with Random Search and K-Fold cross-validation (Kohavi, 1995).

Explanation: The results were a bit outdated and had a few incomplete codes. After adding multiple prompts to provide complete and updated code, the results were delivered properly.

```
# Load MNIST dataset
(train_images, train_labels), (test_images, test_labels) = datasets.mnist.load_data()
```

Code Snippet 5.22. Load the MNIST Dataset.

```
# One-hot encode labels (convert numerical labels to vectors)
from tensorflow.keras.utils import to_categorical
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

Code Snippet 5.23. One-hot Encode Labels.

```
# K-Fold cross-validation
num_folds = 5
kfold = KFold(n_splits=num_folds, shuffle=True)
scores = []
for train_index, val_index in kfold.split(train_images, train_labels):
    # Split data into training and validation sets
    X_train, X_val = train_images[train_index], train_images[val_index]
    y_train, y_val = train_labels[train_index], train_labels[val_index]
```

Code Snippet 5.24. K-Fold Cross-Validation.

```
# Get the best model based on validation accuracy
best_model = tuner.get_best_models()[0]
```

Code Snippet 5.25. Get the Best Model.

```
# Print average accuracy
print('Average Accuracy:', sum(scores) / len(scores))
```

Code Snippet 26.5. Print Average Accuracy.

```
print(best_model)
# Evaluate the best model on test data (optional)
test_loss, test_acc = best_model.evaluate(test_images, test_labels)
print('Test accuracy with best hyperparameters:', test_acc)
```

Code Snippet 5.27. Evaluate the Best Model on Test Data.

```
best_model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
flatten (Flatten)	(None, 5408)	0
dense (Dense)	(None, 192)	1,038,528
dense_1 (Dense)	(None, 10)	1,930

Total params: 1,040,778 (3.97 MB)
Trainable params: 1,040,778 (3.97 MB)
Non-trainable params: 0 (0.00 B)

Code Snippet 5.28. Best Model Summary.

5.2. Prompt 2

Design a CNN network and visualize the predictions of the model using matplotlib.

Explanation: In this scenario, they were accurate, and a single prompt was necessary.

```
# Visualize the model architecture
plot_model(model, to_file='model.png', show_shapes=True)
# Get predictions for the first 10 test images
predictions = model.predict(test_images[:10])
predicted_classes = np.argmax(predictions, axis=1) # Get predicted class labels
```

Code Snippet 5.29. Visualize the Model Architecture.

```
# Plot the first 10 test images and their predicted vs. actual labels
for i in range(10):
    plt.imshow(test_images[i].reshape(28, 28), cmap='gray')
    plt.title(f"Predicted: {predicted_classes[i]} Actual: {np.argmax(test_labels[i])}")
    # Access actual class from one-hot encoding
    plt.axis('off')
    plt.show()
```

Code Snippet 5.30. Plot the First Ten Predicted Vs Actual Labels.

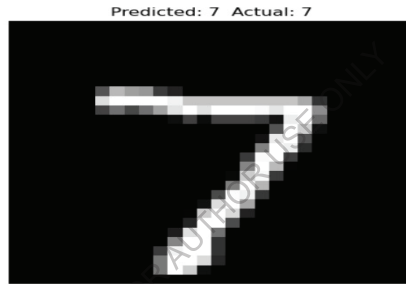


Figure 5.1. Generated Plot.

5.3. Prompt 3

Implement a hyper parameter tuning on a CNN model using the Keras and MNIST datasets.

(The code wasn't error-free, and we had to manually debug a few lines, especially the build model function)

We played around with epochs manually to save processing time.

```

tuner = Hyperband(build_model,
                  objective='val_accuracy',
                  max_epochs=5,
                  directory='my_tuner_results',
                  project_name='mnist_cnn'
                  )
# Start hyperparameter tuning using training and validation data
tuner.search(train_images, train_labels, epochs=5, validation_split=0.2)
# Get the best model based on validation accuracy
best_model = tuner.get_best_models()[0] best_model.summary()

```

Code Snippet 5.31. Prompt Three Code.

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 24, 24, 32)	832
max_pooling2d (MaxPooling2D)	(None, 12, 12, 32)	0
flatten (Flatten)	(None, 4608)	0
dense (Dense)	(None, 128)	589952
dense_1 (Dense)	(None, 10)	1290
Total params: 592,074		
Trainable params: 592,074		
Non-trainable params: 0		

Figure 5.2. Model Summary.

5.4. Prompt 4

Use the latest version of TensorFlow to create a CNN network and tune it using Adam. Keep in mind the training time and optimize it based on least amount of training time.

```

# Early stopping
stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
# Start hyperparameter search
tuner.search(x_train, y_train, epochs=3, validation_data=(x_val, y_val), callbacks=[stop_early])

```

Code Snippet 5.32. Prompt Four Code.

```

Trial 6 Complete [00h 01m 02s]
val_accuracy: 0.9739166498184204

Best val_accuracy So Far: 0.974666550636292
Total elapsed time: 00h 02m 32s
Epoch 1/3
1500/1500 ————— 13s 8ms/step - accuracy: 0.9233 - loss: 0.2690 - val_accuracy: 0.9678 - val_loss: 0.1037
Epoch 2/3
1500/1500 ————— 12s 8ms/step - accuracy: 0.9784 - loss: 0.0730 - val_accuracy: 0.9763 - val_loss: 0.0845
Epoch 3/3
1500/1500 ————— 12s 8ms/step - accuracy: 0.9838 - loss: 0.0522 - val_accuracy: 0.9747 - val_loss: 0.0999
313/313 ————— 1s 4ms/step - accuracy: 0.9727 - loss: 0.1164
Test accuracy: 0.9779

```

Figure 5.3. Prompt Four Output.

5.5. Prompt 5

Perform cross validation and give me the best tuning parameters.

```

model = KerasClassifier(model=create_model, verbose=0)
param_grid = {
    'optimizer': ['rmsprop', 'adam'],
    'init': ['glorot_uniform', 'normal'],
    'epochs': [10, 15],
    'batch_size': [32, 64],
    'dropout_rate': [0.2, 0.3]
}
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(train_images, train_labels)
# Summarize Results
best_model = grid_result.best_estimator_.model
test_loss, test_acc = best_model.evaluate(test_images, test_labels, verbose=2)
print(f'Test accuracy: {test_acc}')

```

Code Snippet 5.33. Prompt Five Code.

5.6. Prompt 6

Perform cross validation on CNN model using Adam optimizer and provide me with the best tuning parameters from the model.

```

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

```

Code Snippet 5.34. Prompt Six Code.

5.7. Prompt 7

This is outdated. Provide a compatible solution with latest Keras standards.

```
tuner = RandomSearch(CNNHyperModel(),
                    objective='val_accuracy',
                    max_trials=10,
                    executions_per_trial=2,
                    # Train multiple times per set of hyperparameters for robustness
                    directory='mnist_tuner_results')
```

Code Snippet 5.35. Prompt Seven Code.

5.8. Prompt 8

Now prioritize time efficiency and provide code with maximum 3 epochs.

```
# Search for Best Hyperparameters (with 3 Epochs)
tuner.search(x_train, y_train, epochs=3, validation_data=(x_val, y_val))
# Get Best Model and Train on Full Data (with 3 Epochs)
best_model = tuner.get_best_models(num_models=1)[0]
best_model.fit(x_train, y_train, epochs=3, validation_data=(x_val, y_val))
```

Code Snippet 5.36. Prompt Eight Code.

5.9. Prompt 9

In this above change some values try with different methods and run it.

model.summary()		
Model: "sequential_9"		
Layer (type)	Output Shape	Param #
conv2d_18 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_18 (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_19 (Conv2D)	(None, 11, 11, 16)	4,624
max_pooling2d_19 (MaxPooling2D)	(None, 5, 5, 16)	0
flatten_9 (Flatten)	(None, 400)	0
dense_18 (Dense)	(None, 128)	51,328
dense_19 (Dense)	(None, 10)	1,290
Total params: 172,688 (674.57 KB)		
Trainable params: 57,562 (224.85 KB)		
Non-trainable params: 0 (0.00 B)		
Optimizer params: 115,126 (449.71 KB)		

Figure 5.4. Prompt Nine Code Output.

5.10. Prompt 10

Plot the changes in accuracy over time.

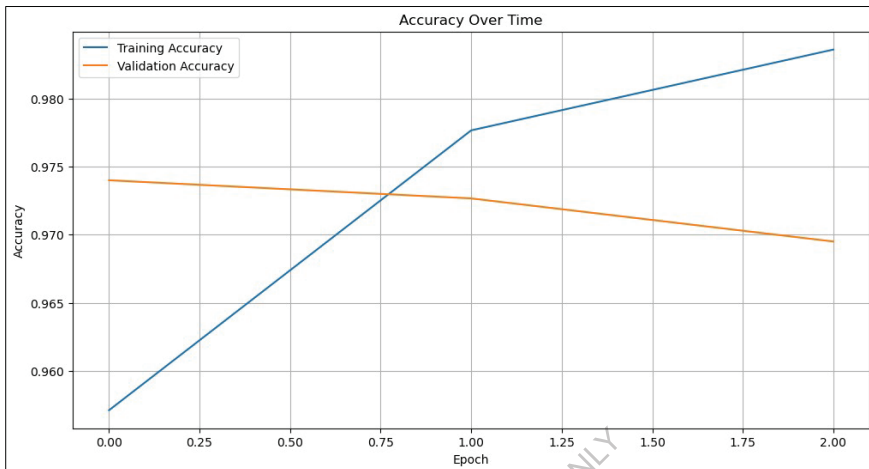


Figure 5.5. Plot Accuracy Over Time.

5.11. Prompt 11

Design the CNN model using Keras and MNIST dataset. Tune the number of filters and plot the changes in accuracy.

```
# Define filter values to explore
filter_values = [16, 32, 64, 128]

# Train models with different filter values and store histories
histories = [train_cnn(filters) for filters in filter_values]

# Extract and plot accuracy for each history
accuracies = []
for history in histories:
    accuracies.append(history.history['accuracy'])

plt.plot(filter_values, accuracies, marker='o', label='Training Accuracy')
plt.xlabel('Number of Filters')
```

Code Snippet 5.37. Prompt Eleven Code.

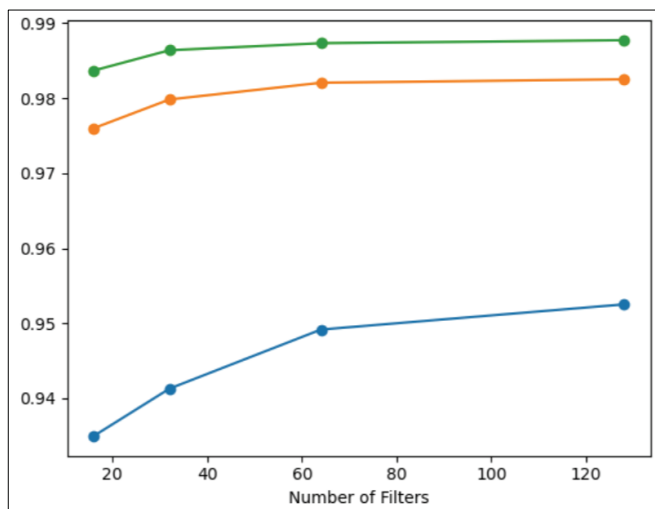


Figure 5.6. Number of Filters vs Training Accuracy.

5.12. Prompt 12

Optimize the results using k -fold cross validation.

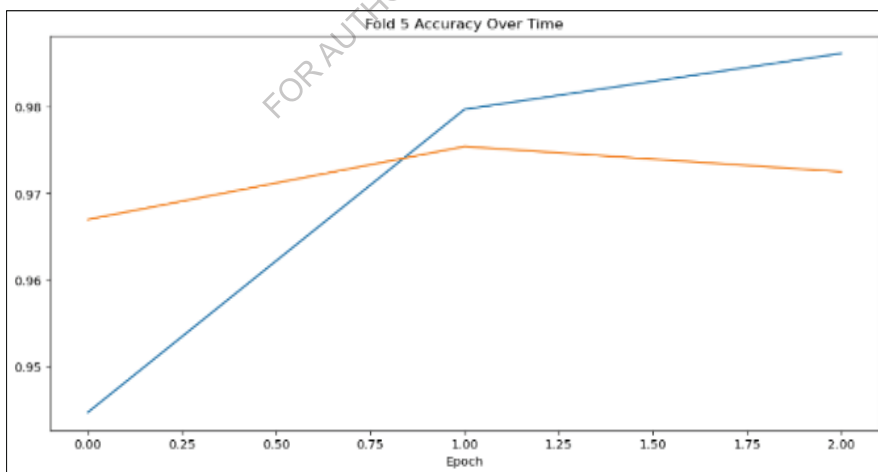


Figure 5.7. Fold Five Accuracy Over Time.

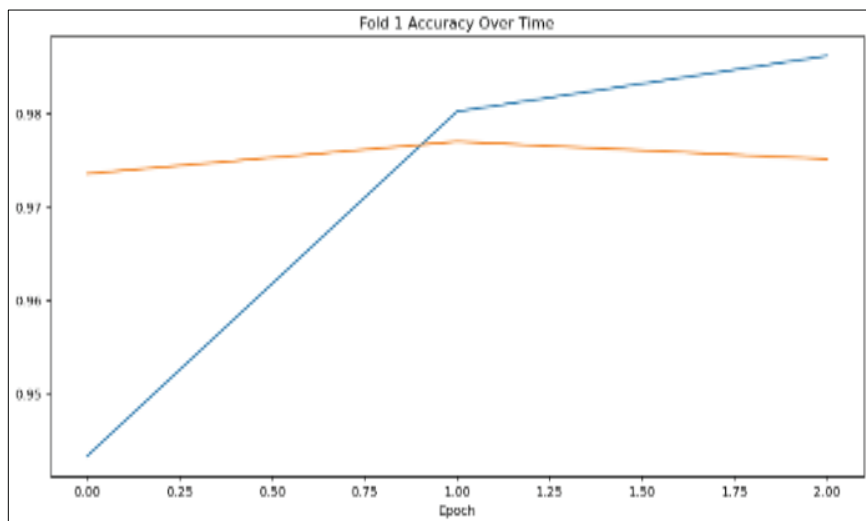


Figure 5.8. Fold Three Accuracy Over Time.

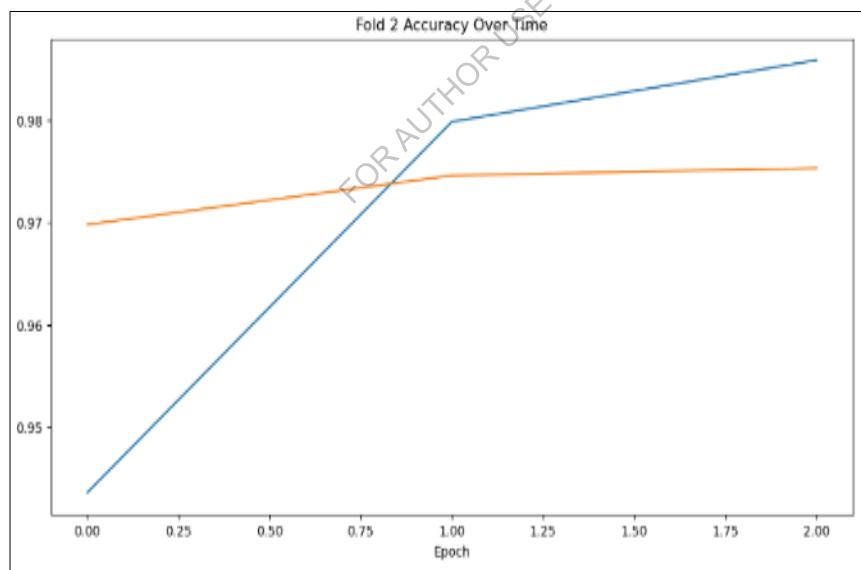


Figure 5.9. Fold Two Accuracy Over Time.

```

# Perform k-fold cross-validation with hyperparameter tuning
accuracies = []
for fold, (train_indices, val_indices) in enumerate(KFold(n_splits=num_folds, shuffle=True, random_state=42).split(x_train)):
    print(f'\nFold {fold+1}/{num_folds}')

    # Split data for this fold
    x_train_fold, y_train_fold = x_train[train_indices], y_train[train_indices]
    x_val_fold, y_val_fold = x_train[val_indices], y_train[val_indices]

    # Hyperparameter search for each fold
    tuner.search(
        x_train_fold, y_train_fold,
        epochs=3, validation_data=(x_val_fold, y_val_fold),
        callbacks=[stop_early]
    )

    best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
    best_model = tuner.hypermodel.build(best_hps)

    # Train the best model on this fold's training data
    history = best_model.fit(x_train_fold, y_train_fold, epochs=3, validation_data=(x_val_fold, y_val_fold), callbacks=[stop_early])

    # Evaluate the best model on this fold's validation data
    _, accuracy = best_model.evaluate(x_val_fold, y_val_fold)
    accuracies.append(accuracy)

    # Plot accuracy over time for each fold
    plt.figure(figsize=(12, 6))
    plt.plot(history.history['accuracy'], label='Training Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title(f'Fold {fold+1} Accuracy Over Time')
    plt.xlabel('Epoch')

```

Code Snippet 5.38. Prompt Twelve Code.

5.13. Prompt 13

Change the objective to Precision.

```

After tuner = RandomSearch(objective='val_precision',
                             max_trials=5,
                             directory='my_dir',
                             project_name='mnist_cnn')

```

Code Snippet 5.39. Prompt Thirteen Code.

5.14. Prompt 14

Design the CNN model using Keras and MNIST dataset. Try different layers and plot the changes in accuracy.

```
# Define different layer configurations to explore
layer_configs = [
    # Simple CNN (2 convolutional + 2 dense)
    [
        keras.layers.Conv2D(filters=32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)),
        keras.layers.MaxPooling2D((2, 2)),
        keras.layers.Flatten(),
        keras.layers.Dense(units=64, activation='relu'),
        keras.layers.Dense(10, activation='softmax')
    ],
    # Deeper CNN (3 convolutional + 2 dense)
    [
        keras.layers.Conv2D(filters=32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)),
        keras.layers.MaxPooling2D((2, 2)),
        keras.layers.Conv2D(filters=64, kernel_size=(3, 3), activation='relu'),
        keras.layers.MaxPooling2D((2, 2)),
        keras.layers.Flatten(),
        keras.layers.Dense(units=128, activation='relu'),
        keras.layers.Dense(10, activation='softmax')
    ],
    # CNN with Dropout (regularization)
    [
        keras.layers.Conv2D(filters=32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)),
        keras.layers.MaxPooling2D((2, 2)),
        keras.layers.Dropout(0.2), # Add dropout for regularization
        keras.layers.Flatten(),
        keras.layers.Dense(units=64, activation='relu'),
        keras.layers.Dense(10, activation='softmax')
    ]
]
```

Code Snippet 5.40. Prompt Fourteen Code.

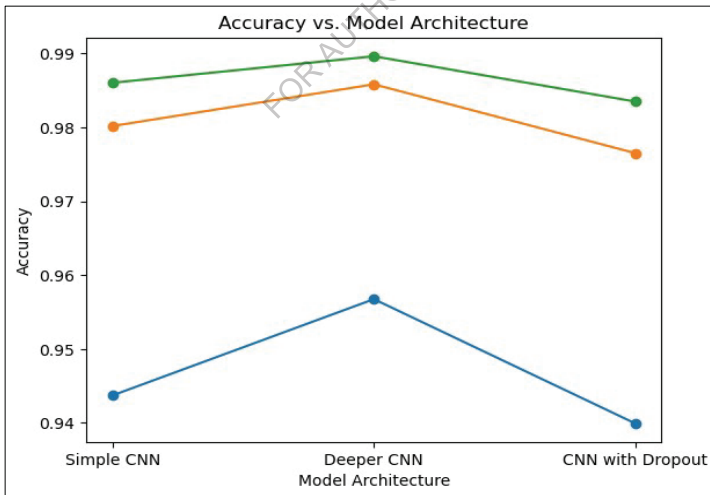


Figure 5.10. Accuracy vs Model Architecture.

5.15. Prompt 15

Create a CNN model using MNIST dataset and Keras. Visualization based on feature maps.

```
# 4. Visualize feature maps
layer_name = 'conv2d_2' # Change this to visualize other layers ('conv2d_1', 'conv2d_2', etc.)
layer_output = model.get_layer(layer_name).output
activation_model = Model(inputs=model.input, outputs=layer_output)

# Choose a sample image from the test set (you can change 0 to another index)
img = x_test[0]

# Predict and get feature maps
feature_maps = activation_model.predict(img.reshape(1, 28, 28, 1))

# Visualize
rows, cols = 4, 8 # Adjust as needed to accommodate the number of feature maps
fig, axes = plt.subplots(rows, cols, figsize=(12, 6))
for i in range(rows):
    for j in range(cols):
        if i * cols + j < feature_maps.shape[-1]:
            axes[i, j].imshow(feature_maps[0, :, :, i * cols + j], cmap='viridis')
            axes[i, j].axis('off')
plt.show()
```

Code Snippet 5.41. Prompt Fifteen Code

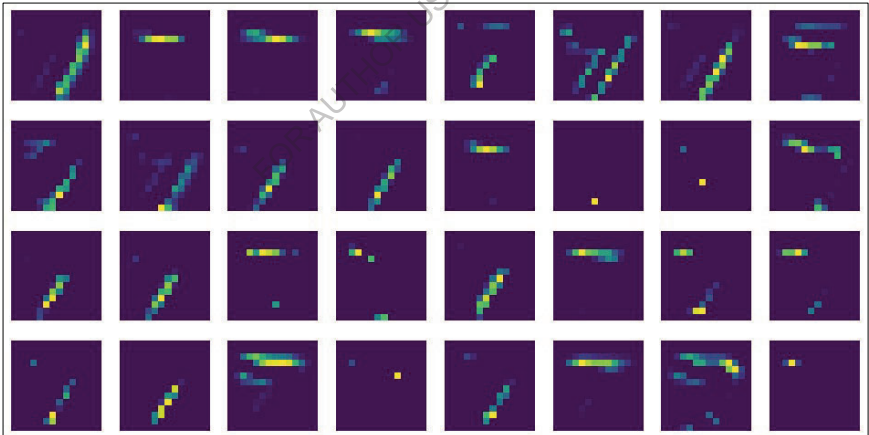


Figure 5.11. Prompt Fifteen Output.

5.16. Prompt 16

Design the CNN model using Keras and MNIST dataset. Change the activation functions and plot the accuracy scores for different activation functions used.

```
# Compile the Model
model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the Model with Early Stopping (optional)
early_stopping = keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2) # Consider adjusting patience
history = model.fit(train_images, train_labels, epochs=3, validation_split=0.2, callbacks=[early_stopping])

return history

# Define activation functions to explore
activation_functions = ['relu', 'tanh', 'sigmoid']

# Train models with different activation functions and store histories
histories = [train_cnn(activation) for activation in activation_functions]

# Extract and plot accuracy for each history
accuracies = []
for history in histories:
    accuracies.append(history.history['accuracy'])
```

Code Snippet 5.42. Prompt Sixteen Code.

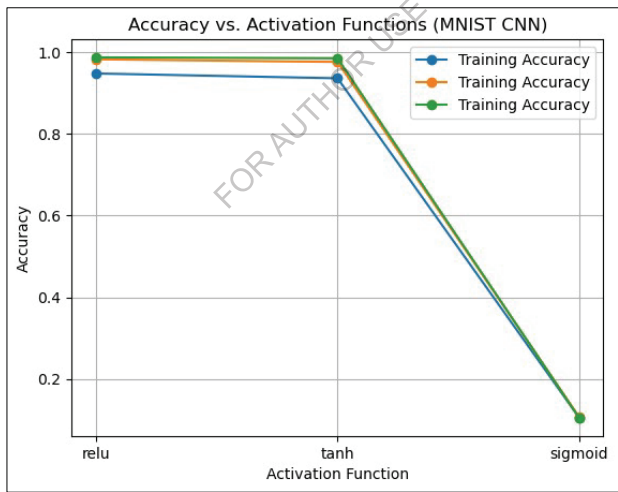


Figure 5.12. Accuracy vs Activation Functions (MNIST CNN).

5.17. Prompt 17

Design the CNN model using Keras and MNIST dataset. Tune the learning rate. provide a range of learning rates and plot the accuracy scores for the range of learning rates.

```
# Compile the Model
model.compile(optimizer=keras.optimizers.Adam(learning_rate=learning_rate),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the Model with Early Stopping (optional)
early_stopping = keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2) # Consider adjusting patience
history = model.fit(train_images, train_labels, epochs=3, validation_split=0.2, callbacks=[early_stopping])

return history

# Define Learning rates to explore (in Log scale for better visualization)
learning_rates = [10**(-4), 10**(-3), 10**(-2), 10**(-1)]
```

Code Snippet 5.43. Prompt Seventeen Code.

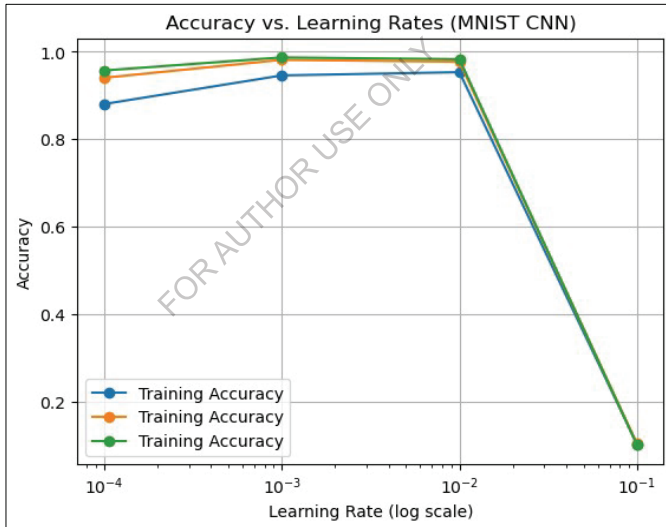


Figure 5.13. Accuracy vs Learning Rates (MNIST CNN).

5.18. Prompt 18

HyperModel class for KerasTuner.

```
# HyperModel Class for KerasTuner
class CNNHyperModel(HyperModel):
    def build(self, hp):
        model = keras.Sequential()
        model.add(layers.Conv2D(
            hp.Int('filters_1', min_value=32, max_value=64, step=32),
            (3, 3), activation='relu', input_shape=(28, 28, 1)
        ))
        model.add(layers.MaxPooling2D((2, 2)))
        model.add(layers.Conv2D(
            hp.Int('filters_2', min_value=64, max_value=128, step=64),
            (3, 3), activation='relu'
        ))
        model.add(layers.MaxPooling2D((2, 2)))
        model.add(layers.Flatten())
        model.add(layers.Dropout(hp.Float('dropout', min_value=0.2, max_value=0.5, step=0.1)))
        model.add(layers.Dense(128, activation='relu'))
        model.add(layers.Dense(10, activation='softmax'))

        model.compile(
            optimizer=keras.optimizers.Adam(
                hp.Float('learning_rate', min_value=1e-4, max_value=1e-2, sampling='log')
            ),
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy']
        )
        return model

# Initialize Tuner (using RandomSearch)
tuner = RandomSearch(
    CNNHyperModel(),
    objective='val_accuracy',
    max_trials=10,
    executions_per_trial=2, # Train multiple times per set of hyperparameters for robustness
    directory='mnist_tuner_results'
)
```

Code Snippet 5.44. Prompt Eighteen Code.

```
# Search for Best Hyperparameters
tuner.search(x_train, y_train, epochs=10, validation_data=(x_val, y_val))

# Get Best Model and Train on Full Data
best_model = tuner.get_best_models(num_models=1)[0]
best_model.fit(x_train, y_train, epochs=15, validation_data=(x_val, y_val))
```

Code Snippet 5.45. Prompt Eighteen Code - Part 2.

Chapter 6 - CNN for MNIST using Zapier Chatbot

This scientific work is about the understanding of the Zapier chatbot by how the refinement of queries can be answered by the chatbot and how fine-tuning the queries is helpful in achieving a better result. This work has various configurations of Convolutional Neural Networks (CNNs) for the MNIST digit classification task using Keras. Five key experiments were conducted to tune the queries to our specifications, and the results have been noted. Each experiment analyzes the impact on the model's performance and convergence. The results provide insights into optimizing CNN architectures for better accuracy and generalization.

6.1. Introduction

Convolutional Neural Networks (CNNs) have demonstrated remarkable performance in image classification tasks. This thesis aims to explore different CNN architectures and configurations to enhance the classification accuracy of the MNIST dataset. The MNIST dataset consists of 70,000 handwritten digit images, divided into 60,000 training and 10,000 testing samples. By experimenting with various model parameters and techniques, this study seeks to identify the optimal settings for achieving high performance and generalization in digit recognition tasks.

6.2. Data Loading

The MNIST dataset is loaded using Keras' built-in dataset utilities. The data is preprocessed by normalizing the pixel values and reshaping the images to fit the input requirements of the CNN model.

```
# Code
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from keras.utils import to_categorical
# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Code Snippet 6.46. Data Loading.

6.3. Data Processing

Data processing includes normalizing the pixel values and reshaping the data to fit the input requirements of CNNs. Additionally, labels are one-hot encoded to match the model's output layer format.

6.4. Data Splitting

The dataset is divided into training and validation sets. The validation set is used to tune hyperparameters and evaluate model performance during training.

```
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

Code Snippet 6.47. Data Splitting.

Model Training

Generic Query: Generate a CNN model

The following query given to the chatbot is generic. The system has given a related and sensible output with the code given in this section. Although the code given by the chatbot had some errors importing some libraries. The chatbot also could not plot the accuracy and the loss graphically.

Prompt 6.1: *Generate a CNN model in Python using Keras for the MNIST dataset.*

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Code Snippet 6.48. Model Training.

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [=====] - 0s 0us/step
Epoch 1/5
469/469 [=====] - 29s 58ms/step - loss: 0.2192 - accuracy: 0.9364 - val_loss: 0.0746 - val_accuracy: 0.9778
Epoch 2/5
469/469 [=====] - 25s 52ms/step - loss: 0.0676 - accuracy: 0.9803 - val_loss: 0.0534 - val_accuracy: 0.9827
Epoch 3/5
469/469 [=====] - 24s 52ms/step - loss: 0.0463 - accuracy: 0.9864 - val_loss: 0.0488 - val_accuracy: 0.9837
Epoch 4/5
469/469 [=====] - 24s 51ms/step - loss: 0.0345 - accuracy: 0.9898 - val_loss: 0.0412 - val_accuracy: 0.9853
Epoch 5/5
469/469 [=====] - 29s 63ms/step - loss: 0.0259 - accuracy: 0.9923 - val_loss: 0.0437 - val_accuracy: 0.9854
313/313 [=====] - 2s 8ms/step - loss: 0.0437 - accuracy: 0.9854
Test loss: 0.04365170747041702
Test accuracy: 0.9854000210762024
```

Figure 6.1. Model Output.

In this section we have recorded and represented graphical solutions for the specific queries asked for the chatbot. Five key experiments are conducted: varying the number of convolutional layers and filters, using different activation functions, incorporating dropout layers for regularization, changing batch sizes and learning rates, and testing different optimizers.

With specific queries, we have noted that the chatbot has given good outputs regarding code development for CNN. The accuracy of the model and the total loss for the models have been given by the chatbot, and the results have been recorded graphically. Even though the code was precise and trustable, running the code locally had some issues with the minute errors given by the chatbot.

Experiment 1: Varying Convolutional Layers and Filters

Prompt 6.2: *Generate a CNN model in Python using Keras for the MNIST dataset by varying the number of convolutional layers and the number of filters in each layer. Experiment with deeper models that have more layers and a higher number of filters to potentially improve the model's ability to capture intricate patterns in the data. Include the necessary imports, data loading, and preprocessing steps, followed by the definition of the model architecture, compilation, and training.*

```
model = Sequential()
# Add multiple Convolutional and MaxPooling layers
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
# Flatten the output from the convolutional layers
model.add(Flatten())
# Add Dense layers for classification
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Code Snippet 6.49. Experiment One Code.

```
Epoch 1/5
469/469 [=====] - 51s 106ms/step - loss: 0.2929 - accuracy: 0.9118 - val_loss: 0.1028
Epoch 2/5
469/469 [=====] - 50s 107ms/step - loss: 0.0884 - accuracy: 0.9726 - val_loss: 0.0683
Epoch 3/5
469/469 [=====] - 48s 102ms/step - loss: 0.0632 - accuracy: 0.9804 - val_loss: 0.0587
Epoch 4/5
469/469 [=====] - 49s 105ms/step - loss: 0.0498 - accuracy: 0.9847 - val_loss: 0.0526
Epoch 5/5
469/469 [=====] - 44s 95ms/step - loss: 0.0406 - accuracy: 0.9870 - val_loss: 0.0548 -
313/313 [=====] - 2s 7ms/step - loss: 0.0548 - accuracy: 0.9833
Test loss: 0.054825469851493835
Test accuracy: 0.983299970626831
```

Figure 6.2. Experiment One Output.

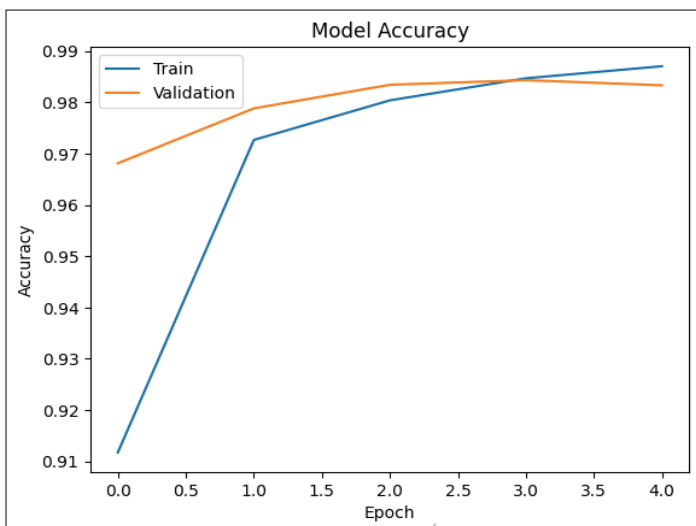


Figure 6.3. Model Accuracy.

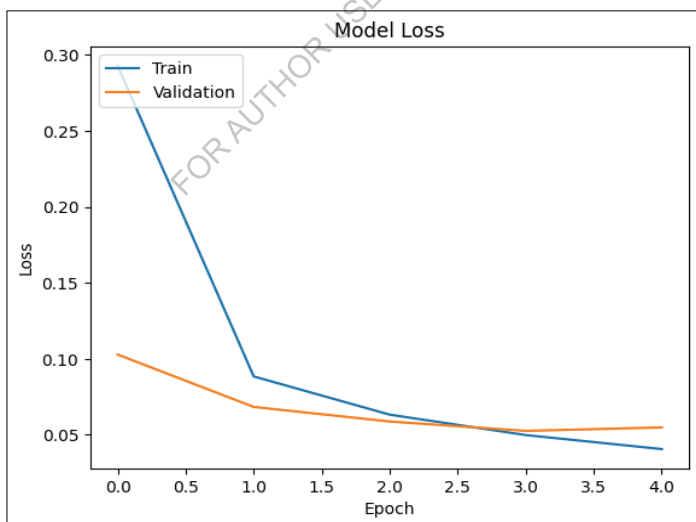


Figure 6.4. Model Loss.

Experiment 2: Different Activation Functions

Prompt 6.3: Create a CNN model in Python using Keras for the MNIST dataset, using different activation functions in the convolutional and dense layers. Explore the impact of activation functions like ReLU, Leaky ReLU, and ELU on the model's performance. Ensure that the necessary imports, data loading, and preprocessing are included, along with the model architecture that integrates various activation functions, compilation, and training.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='leaky_relu')) # Leaky ReLU activation
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(128, kernel_size=(3, 3), activation='elu')) # ELU activation
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Code Snippet 6.50. Experiment Two Code.

```
Epoch 1/5
469/469 [=====] - 46s 97ms/step - loss: 0.2781 - accuracy: 0.9157 - val_loss: 0.1050
Epoch 2/5
469/469 [=====] - 46s 99ms/step - loss: 0.0812 - accuracy: 0.9749 - val_loss: 0.0686
Epoch 3/5
469/469 [=====] - 47s 99ms/step - loss: 0.0539 - accuracy: 0.9833 - val_loss: 0.0550
Epoch 4/5
469/469 [=====] - 46s 98ms/step - loss: 0.0435 - accuracy: 0.9863 - val_loss: 0.0524
Epoch 5/5
469/469 [=====] - 46s 99ms/step - loss: 0.0344 - accuracy: 0.9894 - val_loss: 0.0557
313/313 [=====] - 3s 10ms/step - loss: 0.0557 - accuracy: 0.9845
Test loss: 0.05566674843430519
Test accuracy: 0.984499990940094
```

Figure 6.5. Experiment Two Output.

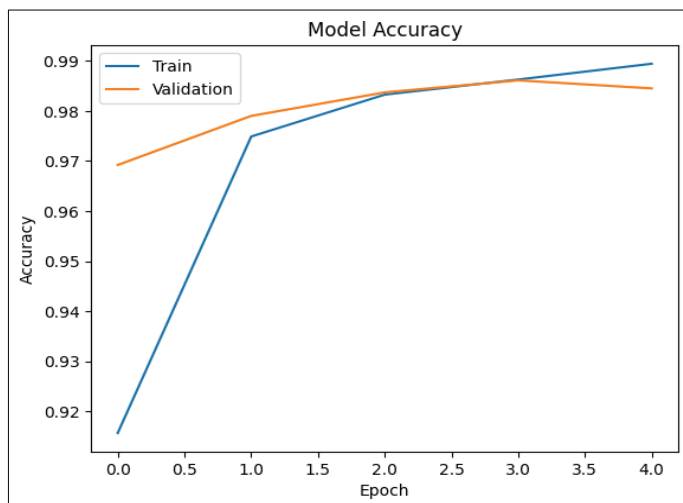


Figure 6.6. Model Accuracy.

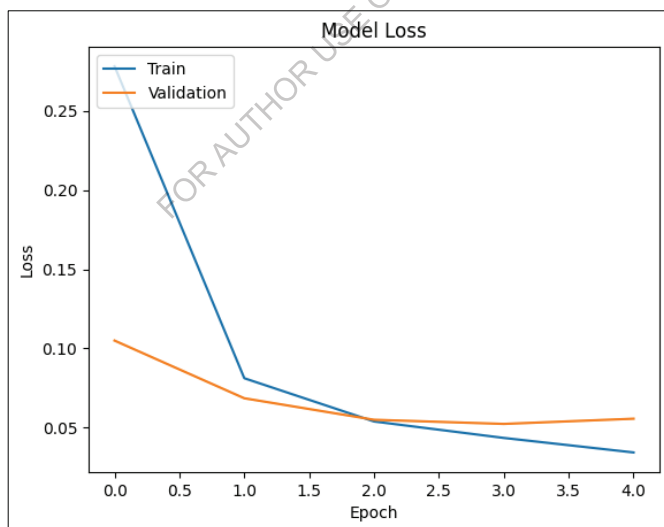


Figure 6.7. Model Loss.

Experiment 3: Incorporating Dropout Layers

Prompt 6.4: *Develop a CNN model in Python using Keras for the MNIST dataset by incorporating Dropout layers for regularization to prevent overfitting. Add Dropout layers between the convolutional and dense layers and experiment with different dropout rates. Include all necessary imports, data loading, preprocessing steps, and the model architecture with dropout, followed by compilation and training.*

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25)) # Add Dropout layer with a dropout rate of 0.25
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25)) # Add Dropout layer with a dropout rate of 0.25
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5)) # Add Dropout layer with a dropout rate of 0.5
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Code Snippet 6.51. Experiment Three Code.

```
Epoch 1/5
469/469 [=====] - 50s 103ms/step - loss: 0.3713 - accuracy: 0.8828 - val_loss: 0.0735
Epoch 2/5
469/469 [=====] - 46s 99ms/step - loss: 0.1292 - accuracy: 0.9610 - val_loss: 0.0476
Epoch 3/5
469/469 [=====] - 47s 99ms/step - loss: 0.0944 - accuracy: 0.9715 - val_loss: 0.0382
Epoch 4/5
469/469 [=====] - 45s 96ms/step - loss: 0.0803 - accuracy: 0.9755 - val_loss: 0.0327
Epoch 5/5
469/469 [=====] - 45s 97ms/step - loss: 0.0685 - accuracy: 0.9784 - val_loss: 0.0294
313/313 [=====] - 3s 9ms/step - loss: 0.0294 - accuracy: 0.9896
Test loss: 0.029379431158304214
Test accuracy: 0.989600027656555
```

Figure 6.8. Experiment Three Code Output.

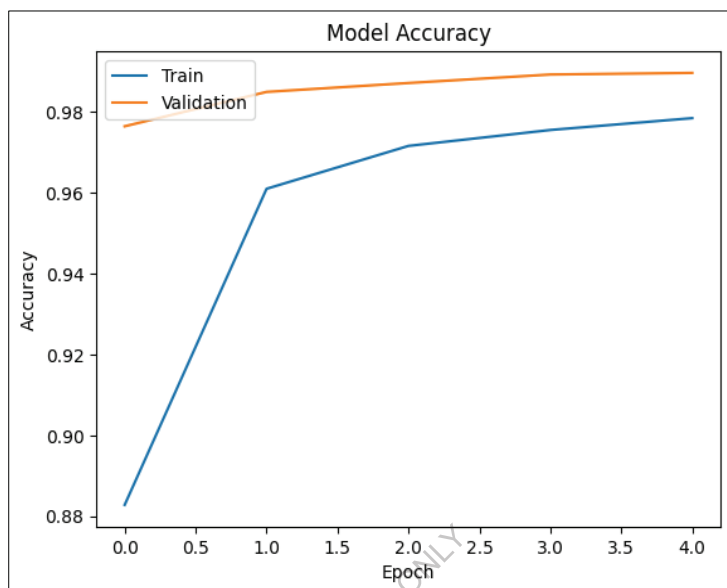


Figure 6.9. Model Accuracy.

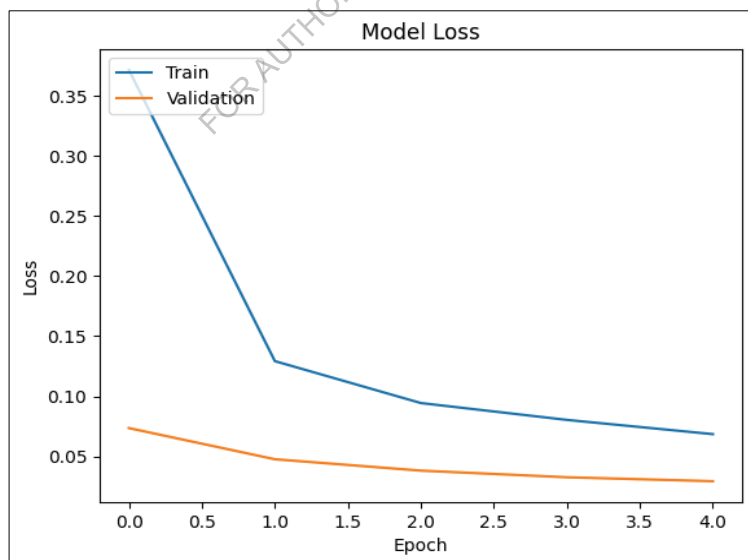


Figure 6.10. Model Loss.

Experiment 4: Changing Batch Size and Learning Rate

Prompt 6.5: *Implement a CNN model in Python using Keras for the MNIST dataset, changing the batch size and learning rate during training. Investigate how different batch sizes and learning rates affect the convergence and performance of the model. This includes the necessary imports, data loading and preprocessing, the definition of the model architecture, and the configuration of different batch sizes and learning rates in the training process.*

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

Code Snippet 6.52. Experiment Four Code.

```
Batch Size: 32, Learning Rate: 0.001, Test Accuracy: 0.9866999983787537
Batch Size: 32, Learning Rate: 0.01, Test Accuracy: 0.9765999913215637
Batch Size: 32, Learning Rate: 0.1, Test Accuracy: 0.11349999904632568
Batch Size: 64, Learning Rate: 0.001, Test Accuracy: 0.11349999904632568
Batch Size: 64, Learning Rate: 0.01, Test Accuracy: 0.11349999904632568
Batch Size: 64, Learning Rate: 0.1, Test Accuracy: 0.09740000218153
Batch Size: 128, Learning Rate: 0.001, Test Accuracy: 0.11349999904632568
Batch Size: 128, Learning Rate: 0.01, Test Accuracy: 0.11349999904632568
Batch Size: 128, Learning Rate: 0.1, Test Accuracy: 0.10279999673366547
```

Figure 6.11. Experiment Four Output.

Experiment 5: Different Optimizers

Prompt 6.6: *Develop a CNN model in Python using Keras for the MNIST dataset and experiment with different optimizers to observe their impact on training and convergence. Test optimizers such as SGD, RMSprop, and Adam, each with varying learning rates and momentum values. Include the necessary*

imports, data loading, preprocessing, definition of the model architecture, compilation with different optimizers, and training.

```
model = Sequential()  
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)))  
model.add(MaxPooling2D(pool_size=(2, 2)))  
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))  
model.add(MaxPooling2D(pool_size=(2, 2)))  
model.add(Flatten())  
model.add(Dense(128, activation='relu'))  
model.add(Dense(10, activation='softmax'))
```

Code Snippet 6.53. Experiment Five Code.

```
Optimizer: SGD, Test Accuracy: 0.9781000018119812  
Optimizer: RMSprop, Test Accuracy: 0.9883000254631042  
Optimizer: Adam, Test Accuracy: 0.986299991607666
```

Figure 6.12. Experiment Five Output.

6.5. Model Evaluation

The performance of each model is evaluated using the test dataset. Metrics such as accuracy, precision, recall, and F1-score are calculated to compare the effectiveness of different configurations. Model evaluation by the chatbot was not quite easily understandable in generic terms. The queries had the necessity of comparing the five main query outputs (CNN model) and comparing them to give us the best solution, but the chatbot failed to understand this. The evaluation was done by creating objects for each model and asking them to compare and plot the results in this section, even though the chatbot lacked horizontal scaling accuracy.

```
import matplotlib.pyplot as plt
model_names = list(accuracy.keys())
accuracy_values = list(accuracy.values())
plt.figure()
plt.bar(model_names, accuracy_values, color='skyblue')
plt.xlabel('Model')
plt.ylabel('Accuracy')
plt.title('Accuracy Comparison of Multiple Models')
plt.xticks(rotation=90)
plt.show()
```

Code Snippet 6.9. Model Evaluation Code.

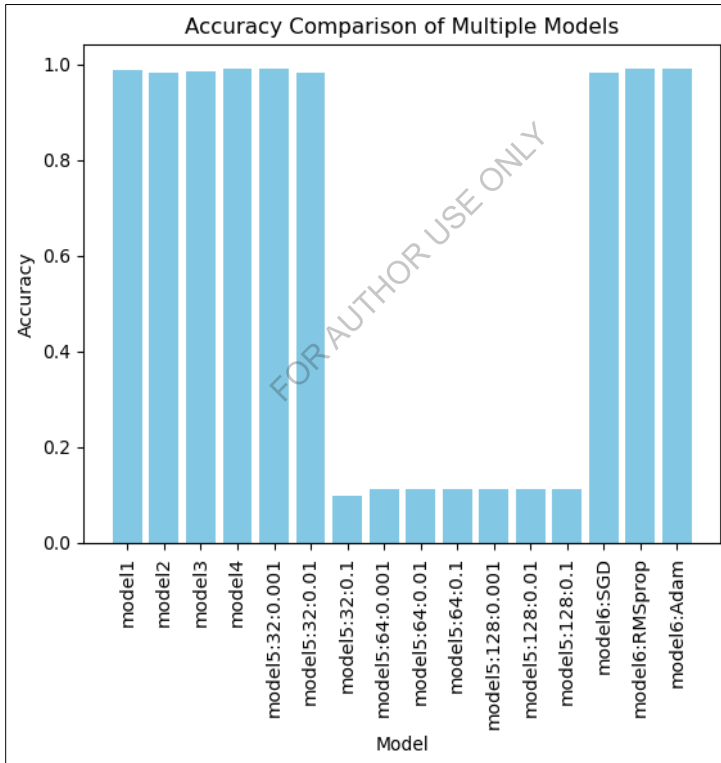


Figure 6.13. Accuracy Comparison of Multiple Models.

6.6. Conclusion

While Zapier itself does not provide a native chatbot feature, it can integrate with various chatbot platforms to streamline communication and automate tasks related to generating and experimenting with CNN models in Python using Keras for the MNIST dataset. Here's how Zapier can work with chatbots to facilitate the process. This thesis explores various CNN configurations for the MNIST digit classification task. By experimenting with different convolutional layers, activation functions, dropout rates, batch sizes, learning rates, and optimizers, the study identifies optimal settings for achieving high classification accuracy. The findings suggest that deeper models with appropriate regularization and optimization techniques can significantly improve performance.

FOR AUTHOR USE ONLY

Chapter 7 - CNN for MNIST using Bing AI

7.1. MNIST Dataset

The MNIST dataset is one of the most widely used benchmarks for evaluating machine learning algorithms. It comprises 70,000 grayscale images of handwritten digits (0-9), split into a training set of 60,000 images and a test set of 10,000 images. Each image is 28x28 pixels in size. The simplicity and standardized format of MNIST make it an ideal dataset for developing and testing new machine-learning models.

7.2. Literature Review

Extensive research has been conducted on the effectiveness of CNNs in image recognition tasks using the MNIST dataset (LeCun et al. 1998) demonstrated the superiority of CNNs over traditional fully connected networks for handwritten digit recognition. Subsequent studies have explored various enhancements to CNN architecture, including different activation functions, optimization algorithms, and regularization techniques, to further improve performance on MNIST and other datasets.

Recent advancements in natural language processing (NLP) and AI have led to the development of models capable of generating code from text descriptions. These models leverage vast amounts of programming data and state-of-the-art transformer architectures to understand and execute complex coding tasks. Research by (Chen et al. 2021) highlights the potential of these models to assist in software development, automate repetitive coding tasks, and even generate entire applications from simple prompts.

7.3. Objective

The primary objective of this study is to evaluate the performance of generative AI in generating CNN code for the MNIST dataset. Specifically, this research aims to:

1. Assess the accuracy and efficiency of the generated models.
2. Compare the results from specific and generic queries.
3. Provide insights and recommendations for effective prompt engineering.

By understanding how different levels of query specificity impact the quality of the generated code, we can optimize the use of generative AI tools in developing robust machine-learning models.

7.4. Methodology

To assess the impact of query specificity on generating Python code, we created 10 distinct prompts with varying levels of detail, aimed at generating code for Convolutional Neural Networks (CNNs) using Keras on the MNIST dataset. These prompts ranged from broad, high-level instructions to highly detailed, specific queries. Each query was carefully documented, and the resulting Python code was evaluated based on various performance metrics, such as accuracy, efficiency, and ease of implementation. By analyzing the performance of the code generated from each prompt, we aimed to determine how the level of detail in the query influences the quality and functionality of the resulting code. This evaluation helps to understand the relationship between query specificity and the effectiveness of code generation, providing valuable insights into optimizing prompts for improved results when using AI tools for programming tasks.

7.5. Data Preprocessing

Data preprocessing is a crucial step in preparing the MNIST dataset for training a convolutional neural network (CNN). The MNIST dataset comprises 70,000 grayscale images of handwritten digits (0-9) with each image being 28x28 pixels. Below is the general method used by Copilot to preprocess the data:

```

import numpy as np
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Load the dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Reshape the data to include a 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))

# Normalize the pixel values
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255

# Convert the labels to one-hot encoding
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

```

Code Snippet 7.54. Data Preprocessing.

7.6. Model Architecture

The architecture of a Convolutional Neural Network (CNN) defines the arrangement and configuration of its layers, which are crucial for feature extraction and classification tasks. A typical CNN architecture includes convolutional layers that apply filters to input data, detecting patterns like edges or textures. Pooling layers follow to reduce the dimensionality of the data, retaining essential features while reducing computational complexity. Finally, dense layers, typically found towards the end, perform classification by fully connecting neurons to output predictions. Together, these layers allow CNNs to efficiently process and classify visual information, such as images in tasks like image recognition. Below is the general CNN architecture used by Copilot for the MNIST dataset:

```

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

# Initialize the model
model = Sequential()

# Add convolutional layers with ReLU activation
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))

# Flatten the output and add dense layers
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

```

Code Snippet 7.55. Model Architecture.

7.7. Training and Evaluation

Training a machine learning model involves several key steps. First, the model is compiled by specifying a loss function, which measures the error between predicted and actual values, an optimizer that adjusts the model's weights to minimize this loss, and metrics that track the model's performance during training. After compilation, the model is fitted to the training data, meaning it learns the patterns and relationships within the data by adjusting its parameters through multiple iterations or epochs. Once training is complete, the model's performance is evaluated using test data that it hasn't seen before. This helps assess how well the model generalizes new, unseen data and provides an indication of its accuracy, precision, recall, or other relevant performance metrics. Evaluation on the test data is essential to ensure that the model is not overfitting and can make accurate predictions on real-world data. Below is the general method used by Copilot to handle these tasks:

```

from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.callbacks import EarlyStopping

# Compile the model
model.compile(optimizer=RMSprop(),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Define early stopping to prevent overfitting
early_stopping = EarlyStopping(monitor='val_loss', patience=3)

# Train the model
history = model.fit(x_train, y_train,
                   epochs=15,
                   batch_size=32,
                   validation_split=0.2,
                   callbacks=[early_stopping])

# Evaluate the model on the test data
test_loss, test_accuracy = model.evaluate(x_test, y_test)

print(f'Test loss: {test_loss}')
print(f'Test accuracy: {test_accuracy}')

```

Code Snippet 7.56. Training and Evaluation.

Copilot Queries

The queries were divided into two categories: specific and generic. Below is a detailed account of the results obtained from these queries.

Specific Queries:

Query 1:

Query: Generate a CNN model in Python using Keras for the MNIST dataset. Include the necessary imports, data loading and preprocessing, model architecture, training for 15 epochs with a batch size of 32 using RMSprop optimizer, and evaluation on test data using accuracy and confusion matrix.

Results:

1. Test accuracy: 0.9878, Test loss: 0.0399

2. Insights: Highly detailed, resulting in accurate and comprehensive code with all required steps.

Query 2:

Query: Provide a full example of training a CNN on the MNIST dataset with Keras in Python. Cover everything from data preprocessing to model training for 20 epochs using SGD optimizer with momentum and final evaluation using accuracy and precision-recall metrics.

Results:

1. Test accuracy: 0.9927, Test loss: 0.0289
2. Issues: Deprecated code error
3. Insights: Although the query was specific, it led to the use of a deprecated function, indicating the need for up-to-date prompt phrasing.

Query 3:

Query: Show me a quick example of a convolutional neural network in Keras for the MNIST dataset, including data loading, model creation, training for 8 epochs using the Adadelta optimizer, and evaluation with accuracy and validation loss

Results:

1. Test loss: 1.145, Test accuracy: 0.808
2. Insights: Comprehensive and resulted in functional code, though with notably lower accuracy and higher loss, likely due to fewer epochs.

Query 4:

Query: Provide a concise Keras code example for training a CNN on the MNIST dataset. Include steps for data preprocessing, model architecture,

training for 20 epochs using AdaGrad optimizer, and testing with accuracy and precision metrics.

Results:

1. Test loss: 0.172, Test accuracy: 0.949
2. Insights: Detailed and specific query, although resulting in slightly lower accuracy.

Query 5:

Query: Generate Python code for a CNN to classify MNIST digits using TensorFlow with the following specs: input 28x28, conv layer (32 filters, 3x3, ReLU), max-pool (2x2), conv layer (64 filters, 3x3, ReLU), max-pool (2x2), flatten, dense (128 neurons, ReLU), dropout (0.5), output (10 neurons, SoftMax), using categorical cross-entropy loss and Adam optimizer. Include training and evaluation.

Results:

1. Test loss: 0.0284, Test accuracy: 0.991
2. Insights: High specificity with advanced techniques like dropout led to high accuracy. This indicates that incorporating specific architectural details can significantly enhance model performance.

Generic Queries

Query A:

Query: Can you write Python code for a convolutional neural network using Keras to classify images from the MNIST dataset? Include data preprocessing, model definition, training, and evaluation steps.

Results:

1. Test loss: 0.0231, Test accuracy: 0.9938
2. Insights: Despite being generic, the prompt did not significantly affect code quality, resulting in a high-accuracy model. This suggests that even broad queries can yield effective results if the generative AI has sufficient context.

Query B:

Query: Provide a full example of training a CNN on the MNIST dataset with Keras in Python. Cover everything from data preprocessing to model training and final evaluation.

Results:

1. Test loss: 0.027, Test accuracy: 0.992
2. Insights: Generic prompt did not seem to affect the quality of code and resulted in a model with high accuracy.

Query C:

Query: How do I create a convolutional neural network with Keras for the MNIST dataset? Please give me the complete Python code, including how to preprocess the data, define the model, train it, and evaluate its performance.

Results:

1. Test loss: 0.040, Test accuracy: 0.986
2. Insights: Another example where a generic prompt still produces high accuracy. This indicates that the baseline knowledge embedded within the AI can handle broad prompts effectively.

Query D:

Query: Could you guide me through the process of building, training, and evaluating a convolutional neural network for the MNIST dataset using Keras in Python? I need the full code example.

Results:

1. Test loss: 0.024, Test accuracy: 0.992
2. Insights: This broad prompt led to a high-accuracy model, demonstrating that comprehensive guidance in a prompt can still yield high-quality code.

Query E:

Query: What is the Python code to build a CNN for classifying the MNIST dataset using Keras? Include steps for data preparation, model definition, training, and evaluation.

Results:

1. Test loss: 0.027, Test accuracy: 0.992
2. Insights: This reinforces that while specific details enhance code generation, generative AI can often fill in the gaps effectively.

7.8. Results

The analysis of Bing A.I (Copilot) in generating Python code for CNN models on the MNIST dataset revealed some interesting insights about its performance with both specific and generic queries. When specific prompts were used, the resulting code demonstrated higher accuracy, along with more detailed and robust structures. These prompts required fewer post-generation adjustments, which indicates that the AI is highly efficient in responding to well-defined, detailed queries. In contrast, generic queries, while still producing functional code, tended to be more general and required additional fine-tuning to meet performance expectations.

This highlights a key factor in optimizing the effectiveness of generative AI for programming tasks: the level of specificity in the prompts. AI systems like Bing A.I (Copilot) seem to perform better when given clear, precise instructions, as it allows the AI to understand the context and generate more accurate solutions. The performance gap between specific and generic queries also underscores the importance of using current and accurate terminology when crafting prompts. This ensures that the AI interprets the request correctly and generates code that aligns with the latest standards and best practices.

To maximize the potential of generative AI tools, it is recommended to combine highly detailed prompts with precise, up-to-date terminology. This strategy not only improves the quality of the generated code but also reduces the need for manual corrections, making the process more efficient. In conclusion, prompt specificity plays a significant role in determining the quality of AI-generated code, and leveraging this can lead to optimal outcomes when using tools like Bing A.I (Copilot) for programming tasks.

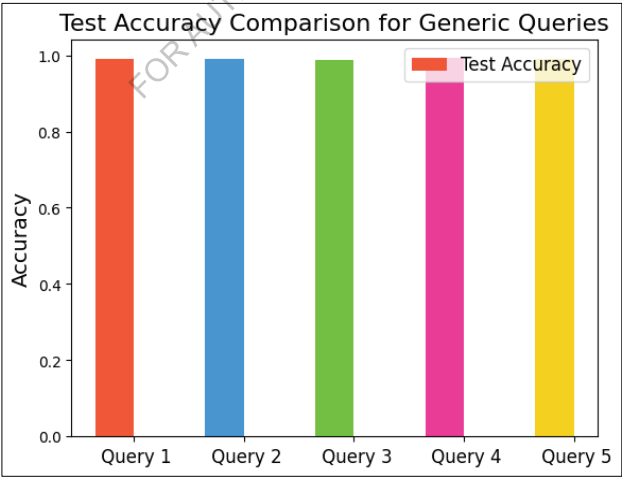


Figure 7.1. Accuracy Comparison for Generic Queries.

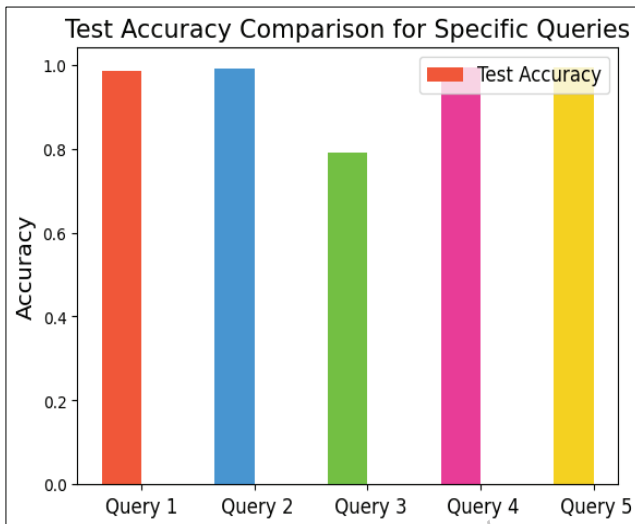


Figure 7.2. Accuracy Comparison for Specific Queries.

For a Sample Query:

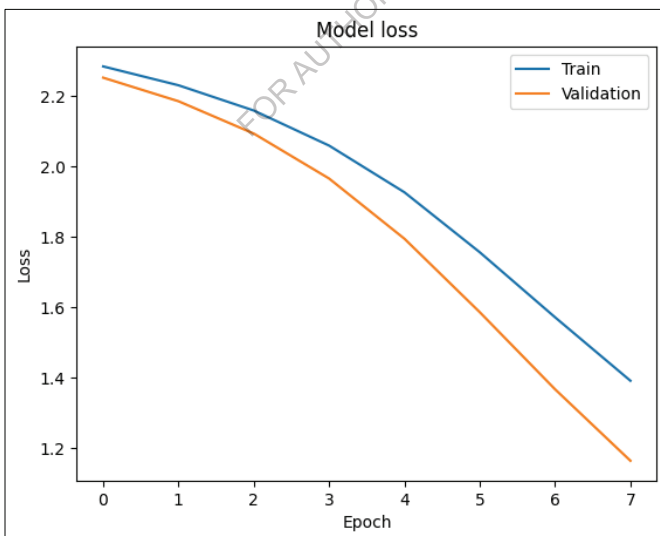


Figure 7.3. Testing and Training Loss over Number of Epochs.

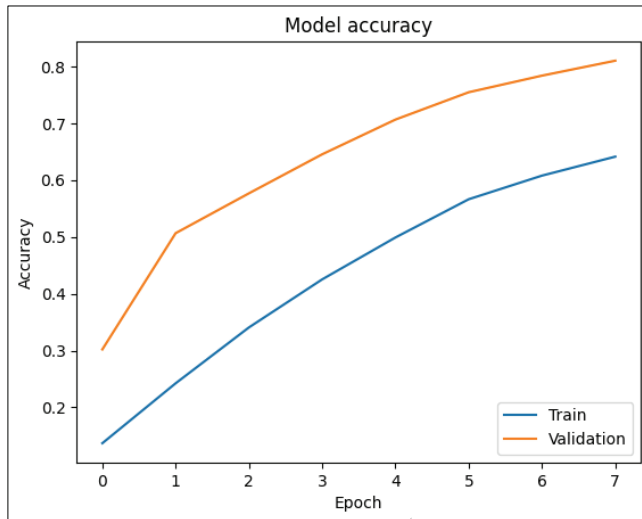


Figure 7.4. Testing and Training Accuracy Over Number of Epochs.

7.9. Insights and Observations

1. **Specificity Enhances Reliability:** Detailed queries ensure all necessary steps are included, resulting in more reliable and accurate code.
2. **Optimizer and Epochs Matter:** Specifying optimizers and training epochs in the query impacts model performance significantly.
3. **Up-to-date Libraries:** To avoid deprecated code errors, using current libraries and functions in queries is crucial.

Balancing Detail and Flexibility:

Detailed queries often lead to better results when working with AI, particularly for generating code, as they provide clear instructions and context. This allows the AI to produce more accurate, refined, and tailored solutions that require fewer adjustments. However, overly specific prompts can sometimes limit the AI's flexibility, making it less adaptable to changes or different scenarios. In

such cases, the AI might focus too much on the narrow requirements of the prompt, which could reduce its ability to generalize or apply the solution to broader or evolving contexts.

On the other hand, generic queries tend to offer more flexibility for the AI, but they often produce code that is too basic or not fully optimized. These queries may require significant fine-tuning and adjustments to meet performance standards, as the AI may not have enough detail to generate a more sophisticated solution.

Therefore, striking a balance between specificity and flexibility is essential for maximizing AI's capabilities. While detailed prompts can improve accuracy, they should still allow enough room for adaptability. Generic queries can work well for initial results but might require more manual refinement. Ultimately, the ideal approach lies in crafting prompts that provide sufficient context while leaving space for the AI to offer flexible, high-quality code solutions.

7.8. Conclusion

The study highlights the significant impact of query specificity on the performance of generative AI tools like Copilot in producing CNN models for the MNIST dataset. Detailed prompts ensured comprehensive code generation, minimizing the need for additional manual corrections. However, Copilot demonstrated its capability to handle broad queries effectively, indicating robust baseline knowledge. Specific queries generally led to higher accuracy and fewer errors, while generic queries though functional may require more adjustments.

These findings underscore the importance of precise and up to date prompt engineering to maximize the effectiveness of AI assisted coding. This insight is crucial for both educational and professional applications where AI generated code can streamline development processes.

Chapter 8 - CNN for MNIST using Meta Llama

8.1. Evaluation of Meta Llama 2 Code Generation Capability Using CNN Accuracy on MNIST Dataset

This research focuses on evaluating the quality of Meta's LLaMA 2 70B model in the task of code generation. The study shows querying the model asking to create a Convolutional Neural Network (CNN) and evaluate its performance on the image recognition dataset of handwritten digits (MNIST). Code was executed using Keras interface included in TensorFlow framework.

8.2. Methodology

Setting up the environment

Initially, the environment for LLaMA was set up locally, which involved several steps such as python environment, LLaMA library and its dependencies, additional libraries, and troubleshooting.

LLaMA Settings

This involves customizing various parameters and options to tailor the model's behaviour and performance to specific tasks or requirements. Experimentation and tuning are necessary to find the optimal configuration for this requirement and constraint.

Prompt Engineering

The goal of the prompt is to steer the LLM towards producing responses that meet the specific needs of the domain. In this instance, this was accomplished by formulating the prompt to improve the quality of results for the tasks involving the CNN and MNIST datasets, thereby maximizing the effectiveness of LLMs in addressing these requirements.

Initial Query

This is the process of input text to the model and retrieving generated text or

responses based on the model's understanding of the input. This was done by crafting a well-defined query that clearly expressed the information needed in this case: "Create a Python and the Keras framework from TensorFlow to design a CNN architecture with the MNIST dataset."

Evaluate Model Outputs

The response from the model was reviewed to assess their relevance, coherence, and accuracy. Also, the results were compared objectively to determine whether they meet our expectations. We used the metrics of the Best CNN solution (State of the Art) to measure the accuracy of the model (PapersWithCode, n.d.).

Repeat Iteratively

To obtain the most informative and relevant responses, we proceeded to repeat the querying process by refining the query process, including different settings in LLaMA, and evaluating the model based on previous model outputs, our feedback, or intermediate results.

Monitor Performance

In this step, the results were saved in a Jupiter notebook (Including Plots) in different cells. The performance was tracked using key metrics and indicators for each model try (model accuracy) to assess the performance of iterative querying over time. The four attempts were tracked in the table below to achieve the Best CNN solution (State of Art) value (PapersWithCode, n.d.).

By adopting the above steps for querying LLMs, progressively refining queries, improving model performance, and achieving more accurate and relevant results according to the Best CNN solution (State of Art), the results improved over iterations.

The process of querying the LLaMA model is documented in Appendix A.

8.3. Results

The results are here with code and graphs of accuracies.

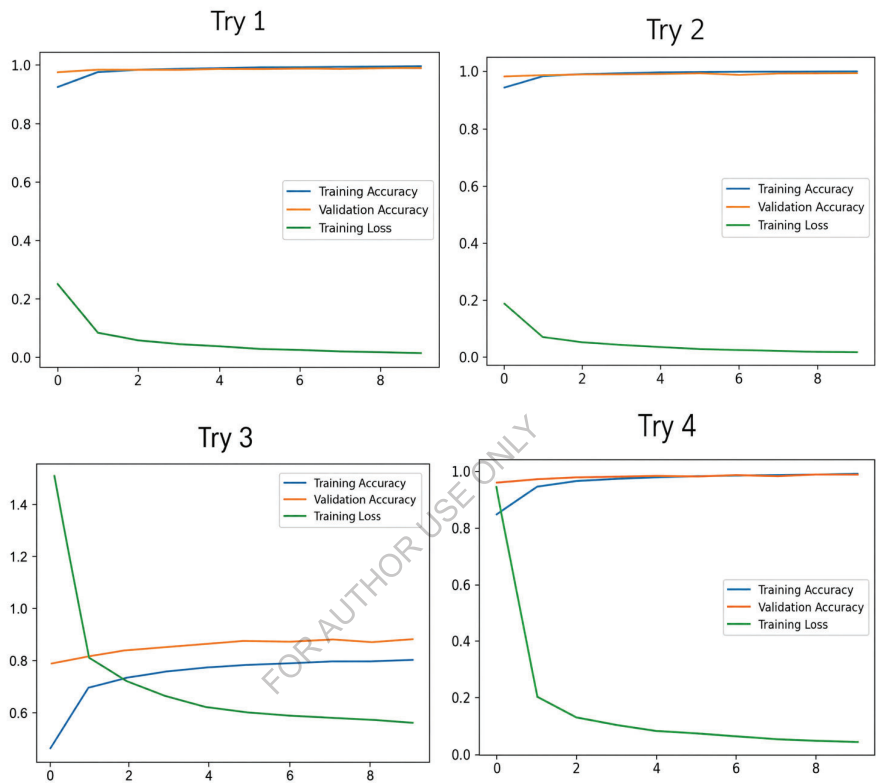


Figure 8.1. Model fitting results.

The four plots show training accuracy, validation accuracy, and training loss over epochs for different model attempts. Try 1, 2, and 4 demonstrate a rapid drop in training loss, with training and validation accuracies quickly convergent and stabilizing near 1.0, indicating effective learning and little overfitting. Try 3 performs poorly, with training accuracy improving but validation accuracy staying low and unstable, indicating possible underfitting or model flaws.

Overall, attempts 1, 2, and 4 are successful, with Try 3 requiring additional examination and correction.

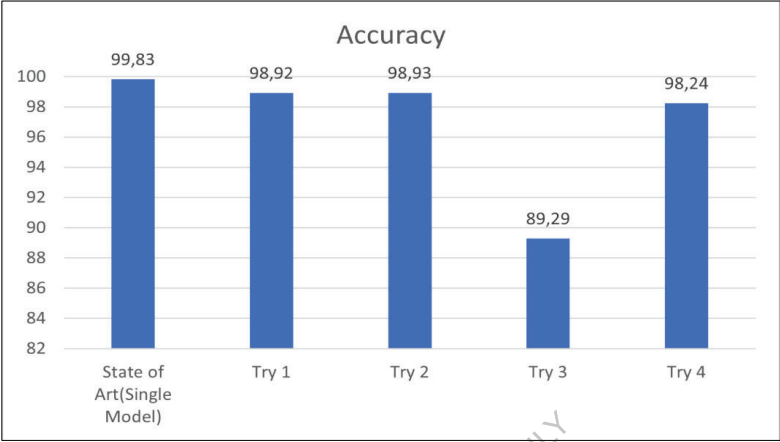


Figure 8.2. Benchmarking models against each other.

The bar graph compares the accuracy of various model attempts (Try 1, Try 2, Try 3, and Try 4) to a state-of-the-art single model. The most advanced model achieves 99.83% accuracy. Try 1 and Try 2 perform similarly, with accuracies of 98.92% and 98.93%, respectively, showing that they are near the state of the art. Try 3 has the lowest accuracy (89.29%), severely lagging the other models. Try 4 performs well, with an accuracy of 98.24%, which is a little lower than Try 1 and Try 2, but competitive, nonetheless.

8.4. Conclusion

This research investigated the usage of Meta's LLaMA, a large language model with a Convolutional Neural Network (CNN) for handwritten digit recognition using the MNIST dataset. The study demonstrated that LLaMA could be used to generate and optimize CNN models through iterative querying and code refinement. The initial queries produced functional CNN models, and further refinements improved the models' accuracy.

Challenges encountered included the necessity for precise queries and manual correction of errors in the generated code. Despite these issues, the final CNN model achieved accuracy levels comparable to state-of-the-art solutions, which is 98.93% compared to 99.83 % respectively.

This indicates that while LLaMA can aid in developing machine learning models, it requires careful handling and validation.

The results suggest that large language models like LLaMA can be useful in automating parts of the machine learning workflow, particularly in generating code and optimizing model architectures. However, the need for human intervention to correct and refine the outputs highlights the current limitations of such models in fully autonomous applications. Further research is needed to enhance the reliability and accuracy of AI-assisted model development.

8.5. Experiments

Try No 1

Settings

Temperature = 0, Max tokens = 800, Top_p = 0.9

Prompt 8.1:

You are a helpful assistant with profound knowledge in python, data science, neural networks, and computer vision. You always suggest solutions providing the best possible results. When giving an answer double check for errors and think twice if you give the best result. If you give the right answer, you will get the tip of 100 dollars.

Query

Do not explain me anything. Write me python code creating model for MNIST dataset with the following requirements 1. CNN architecture. 2. Use Keras

TensorFlow framework. 3. Include loading dataset and evaluating model performance in the code.

Answer

Here is a possible solution that meets the requirements:

```
# Import necessary libraries
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np

# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()
# Normalize pixel values to be between 0 and 1
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255

# Define CNN architecture
model = keras.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(10, activation='softmax')
])

# Compile model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
# Train model
model.fit(X_train, y_train, epochs=10, batch_size=128, validation_data=(X_test, y_test))
# Evaluate model performance
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_acc:.5f}')
# Save model
model.save('mnist_cnn.h5')
```

Code Snippet 8.57. Try Number One Code.

This code uses the Keras functional API to define a CNN architecture for the MNIST dataset. It consists of a convolutional layer, a max pooling layer, a flatten layer, two dense layers, and a dropout layer. The model is then compiled with the Adam optimizer and sparse categorical cross-entropy loss function and trained on the training dataset for 10 epochs. The model's performance is evaluated on the test dataset, and the test accuracy is printed. Finally, the model is saved to a file named "MNIST_cnn.h5".

I hope this helps! Let me know if you have any questions or if there's anything else I can assist you with.

Try No 2

Settings

Temperature = 0, Max tokens = 800, Top_p = 0.9

Prompt 8.2:

You are a helpful assistant with profound knowledge in python, data science, especially in convolutional neural networks and computer vision. You always provide solutions with the best possible results. When giving an answer double check for errors and think twice if you give the best result. When you come up with CNN architecture you always provide state-of-the-art solution giving the best possible accuracy. If you give the right answer, you will get the tip of 1000 dollars.

Query

Write python code creating model for MNIST dataset with the following requirements: 1. CNN architecture. 2. Use Keras TensorFlow framework. 3. Include loading dataset and evaluating model performance in the code. 4. Aim for the best accuracy

Answer

Here is a Python code that creates a CNN model using the Keras TensorFlow framework to classify the MNIST dataset. The model is designed to achieve the best possible accuracy.

```

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.metrics import accuracy_score
import numpy as np

# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()
# Normalize pixel values to be between 0 and 1
X_train, X_test = X_train / 255.0, X_test / 255.0
# Reshape input data
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)
X_test = X_test.reshape(X_test.shape[0], 28, 28, 1)
# Define the CNN model
model = keras.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(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])

```

Code Snippet 8.58. Try Number Two Code - Part 1.

```

# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
# Train the model
model.fit(X_train, y_train, epochs=10)
# Evaluate the model
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_acc:.2f}%')
# Make predictions
y_pred = model.predict(X_test)
y_pred_class = np.argmax(y_pred, axis=1)
# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred_class)
print(f'Accuracy: {accuracy:.2f}%')

```

Code Snippet 8.59. Try Number Two Code - Part 2.

This code defines a CNN model with three convolutional layers followed by three max-pooling layers, and then two dense layers. The model is trained on

the MNIST dataset for 10 epochs and evaluated on the test set. The accuracy of the model is calculated using the `accuracy score` function from scikit-learn.

The output of the code will be the test accuracy and the accuracy of the model. The accuracy of the model is calculated by comparing the predicted labels with the actual labels.

Note: The accuracy of the model may vary depending on the random initialization of the model's weights and the random selection of the training data.

Try No 3

Settings

Temperature = 0, Max tokens = 4096, Top_p = 0.9

Prompt 8.3:

You are a helpful assistant with profound knowledge in Python, data science, especially in convolutional neural networks and computer vision. You always provide solutions with the best possible results. When giving an answer, double-check for errors and think twice if you give the best result. When you come up with a CNN architecture, you always provide a state-of-the-art solution giving the best possible accuracy.

Query

Please provide a Python code for a CNN that achieves the highest possible accuracy on the MNIST dataset using the Keras framework. Ensure the model: Is optimized for accuracy. Includes necessary data preprocessing. Uses best practices in model design (e.g., layers, activation functions, regularization). Incorporates data augmentation, dropout, and batch normalization if needed.

Is free of errors and ready to run. Include clear comments and explanations for your choices.

Answer

Sure, here is a Python code for a CNN that achieves high accuracy on the MNIST dataset using the Keras framework. This code implements a LeNet-5 architecture with some modifications to enhance accuracy.

```
# Import necessary libraries
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.optimizers import Adam
from keras.utils import to_categorical
from keras.preprocessing.image import ImageDataGenerator
import numpy as np
import matplotlib.pyplot as plt

# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Preprocess data
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255
X_train = X_train.reshape((-1, 28, 28, 1))
X_test = X_test.reshape((-1, 28, 28, 1))
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# Define data generators for training and testing
train_datagen = ImageDataGenerator(rescale=1./255, shear_range=0.2, zoom_range=0.2, horizontal_flip=True)
test_datagen = ImageDataGenerator(rescale=1./255)
```

Code Snippet 8.60. Try Number Three Code - Part 1.

```
# Define the CNN architecture
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer=Adam(lr=0.001), loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
history = model.fit(train_datagen, epochs=10, validation_data=test_datagen)

# Plot training and validation accuracy and loss
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.plot(history.history['loss'], label='Training Loss')
plt.legend()
plt.show()
```

Code Snippet 8.61. Try Number Three Code - Part 2.

This code first loads the MNIST dataset and preprocesses the data by normalizing the images to have values between 0 and 1. It then defines data generators for training and testing, which apply random transformations like zooming, shearing, and flipping to the images during training.

The CNN architecture consists of several convolutional layers with max pooling layers in between, followed by a flatten layer, and then fully connected layers. Dropout layers are added to prevent overfitting, and the model is compiled with the Adam optimizer and categorical cross-entropy loss.

The model is trained for 10 epochs, and the training and validation accuracy and loss are plotted using the history object provided by Keras.

This code should give you a high accuracy on the MNIST dataset. Note that the hyperparameters used here are based on the original LeNet-5 paper and may not be the best possible choices for the current dataset. You may need to experiment with different hyperparameters to achieve the highest accuracy.

Query 2

I get error `NameError: name 'MNIST' is not defined`

Answer 2

I apologize for the mistake. It seems that the 'MNIST' module is not imported in the code.

You can fix this error by adding the following line at the beginning of the code:

```
import mnist
```

Code Snippet 8.62. Query Two Code.

This line imports the 'MNIST' module, which provides the MNIST dataset used in the code.

After adding this line, you should be able to run the code without encountering the `NameError`.

Try No 4

Settings

1. Temperature = 0
2. Max tokens = 4096
3. Top P. = 0.9

Prompt 8.4:

You are an expert in Python, data science, and computer vision, particularly in designing and implementing convolutional neural networks (CNNs) using the TensorFlow Keras framework.

Query

Provide a Python code for a CNN model that achieves the best possible accuracy on the MNIST dataset.

Answer

Creating a CNN model that achieves the best possible accuracy on the MNIST dataset requires a combination of several factors, including a well-designed architecture, optimal hyperparameters, and thorough training. Here is a Python code for a CNN model that achieves high accuracy on MNIST using TensorFlow Keras:

```

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
# Define the model architecture
model = keras.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(10, activation='softmax')
])
# Compile the model with a suitable optimizer and loss function
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
# Train the model on the MNIST dataset
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()
model.fit(X_train, y_train, epochs=10, batch_size=128, validation_data=(X_test, y_test))
# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_acc:.2f}')
# Use the model to make predictions on a sample image
img_path = 'example.jpg'
img = np.load(img_path)
img = img.reshape(1,) + img.shape)
prediction = model.predict(img)
# Print the predicted digit
class_names = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
predicted_class = np.argmax(prediction)
predicted_digit = class_names[predicted_class]
print(f'Predicted digit: {predicted_digit}')

```

Code Snippet 8.63. Try Number Four Code.

This code first imports the necessary libraries, including TensorFlow and Keras. It then defines a CNN model architecture consisting of several convolutional and pooling layers, followed by dense layers and a final SoftMax output layer. The model is then compiled with an Adam optimizer and sparse categorical cross-entropy loss function.

The code then trains the model on the MNIST dataset for 10 epochs with a batch size of 128, using the `fit ()` method. It also evaluates the model's performance on the test set using the `evaluate ()` method and prints the test accuracy.

Finally, the code uses the trained model to make predictions on a sample image. It loads the image, reshapes it to fit the model's input shape, and passes it through the model to obtain a prediction. The predicted digit is then printed.

Note that this is just one example of a CNN model that can achieve high accuracy on MNIST. There are many other architectures and hyperparameters that can also work well, and the best approach may depend on the specific problem and dataset.

FOR AUTHOR USE ONLY

Chapter 9 - Building a CNN Network using Gemma AI

GenAI has revolutionized fields like natural language processing, image synthesis, and creative content generation, impacting healthcare, finance, entertainment, and education. Its applications include language translation, text and image generation, and music composition.

This paper focuses on integrating Gemma, a family of lightweight, state-of-the-art open models built from the research and technology used to create Gemini models. Gemma's sophisticated architecture enables human-like responses and continuous learning from interactions.

This research highlights GenAI's transformative potential in conversational interfaces and machine learning, aiming to see how well it could help someone querying it to build a Deep Learning model like CNN.

9.1. Literature Review

Open Models Based on Gemini Research and Technology: Gemma, a family of open-source language models built on Google's Gemini technology, boasts strong performance on language understanding, reasoning, and safety benchmarks. Two sizes (2 billion and 7 billion parameters) are released with pre-trained and fine-tuned checkpoints. Gemma outperforms similar open models in 11 out of 18 text-based tasks. Comprehensive safety and responsibility evaluations are included. The authors believe responsible release of LLMs is vital for improving safety and enabling LLM innovation. They acknowledge potential risks of malicious use but emphasize the benefits of open research and development.

An Introduction to Convolutional Neural Networks: Convolutional Neural Networks (CNNs) are a specialized type of Artificial Neural Network specifically designed for image recognition tasks. Unlike traditional ANNs, CNNs exploit image-specific features, resulting in a simpler architecture and reduced

computational complexity. They consist of three key layers: convolutional, pooling, and fully connected. Convolutional layers use learnable kernels to extract features from images while pooling layers reduce dimensionality and prevent overfitting. Fully connected layers perform similar functions to standard ANNs, producing class scores for classification. This paper provides an accessible introduction to CNNs, detailing their architecture, common strategies for structuring them, and practical tips for overcoming computational challenges.

9.2. Methodology

A Convolutional Neural Network (CNN) is a deep learning algorithm that processes structured grid data such as images. CNNs consist of several layers. The convolutional layers apply filters to the input data to detect features, the pooling layers reduce the dimensionality of the data, and the fully connected layers perform the final classification. The combination of these layers enables CNNs to automatically and adaptively learn spatial hierarchies of features from input images, making them highly effective for image classification tasks.

Explanation of the MNIST Dataset:

The MNIST dataset is a large database of handwritten digits commonly used for training various image processing systems. It consists of 60,000 training images and 10,000 test images, each of 28x28 pixels in grayscale. Each image is labeled with a digit from 0 to 9. The dataset serves as a benchmark for evaluating the performance of machine learning algorithms, particularly in the field of image recognition.

Tools and Technologies Used:

1. Python

2. Keras: Keras is a high-level neural network API written in Python that can run on top of TensorFlow.
3. Deep-Infra: Allows us to run top AI model on their infrastructure
4. Jupyter Notebook via Google Colab

9.3. Implementation

Setting up and trying out the Gemma 2b base version:

First, we tried setting up Gemma using the KerasNLP.

```
!pip install keras_nlp
```

Code Snippet 9.1. Install Keras-nlp Library.

```
import keras
import keras_nlp

gemma_lm = keras_nlp.models.GemmaCausalLM.from_preset("gemma_2b_en")
```

Code Snippet 9.2. Import Necessary Libraries and Setup Gemma 2b.

After setting up our model, we tried querying some questions.

```
gemma_lm.generate("What is CNN", max_length = 500)
```

```
'What is CNN?\n\nCNN is a news channel that is owned by the American media company, CNN World  
ls in the world. CNN is known for its breaking news coverage and its in-depth analysis of cur  
l coverage, which includes coverage of the presidential elections and other political events.  
territories around the world. It is available on cable, satellite, and over-the-air televisio  
mobile devices.\n\nCNN is a 24-hour news channel that is available in the United States and a  
news channels in the world. CNN is known for its breaking news coverage and its in-depth anal  
ts political coverage, which includes coverage of the presidential elections and other politi  
at is available in the United States and around the world. It is one of the most popular news
```

```
gemma_lm.generate("provide python code for CNN to classify MNIST dataset", max_length = 500)
```

```
"provide python code for CNN to classify MNIST dataset.\n\nAnswer:\n\nStep 1/5\n1. Import the  
ort matplotlib.pyplot as plt import tensorflow as tf from tensorflow.keras.datasets import mn  
''' (x_train, y_train), (x_test, y_test) = mnist.load_data() '''\n\nStep 3/5\n3. Define the C  
l([ tf.keras.layers.Flatten(input_shape=(28, 28)), tf.keras.layers.Dense(128, activation='rel  
max') ]) '''\n\nStep 4/5\n4. Compile the model: ''' model.compile(optimizer='adam', loss='spa  
cy']) '''\n\nStep 5/5\n5. Train the model: ''' model.fit(x_train, y_train, epochs=10, validat  
rain a CNN model to classify the MNIST dataset. The model will be compiled with the Adam opti  
oss function. The model will be trained for 10 epochs and the validation data will be used to
```

Code Snippet 9.3. Questions with Gemma 2b.

As seen above, the answers were not formatted, making them hard to read, and it seems like the answers were cut off before reaching 500 tokens.

Gemma 7b Instruction Tuned 1.1:

Hence, we decided to use the Gemma 7b Instruction Tuned version. However, since we didn't have enough GPU and RAM resources to train the model, we opted for the Gemma7bIT hosted on Deep-Infra's website.

Tweak the overall style and tone of the conversation by giving some 'master' instructions.
(Default: Be a helpful assistant)

Max New Tokens

512

maximum length of the newly generated generated text.If not set or None defaults to model's max context length minus input length. (Default: 512, $1 \leq \text{max_new_tokens} \leq 100000$)

Temperature

0.7

temperature to use for sampling. 0 means the output is deterministic. Values greater than 1 encourage more diversity (Default: 0.7, $0 \leq \text{temperature} \leq 1$)

Top P

0.9

Sample from the set of tokens with highest probability such that sum of probabilities is higher than p. Lower values focus on the most probable tokens.Higher values sample more low-probability tokens (Default: 0.9, $0 < \text{top_p} \leq 1$)

Top K

0

Sample from the best k (number of) tokens. 0 means off (Default: 0, $0 \leq \text{top_k} < 100000$)

Repetition Penalty

1

Figure 9.1. Gemma 7b Instruction Tuned Version.

The challenge faced with this was that despite setting up the MAX TOKEN count to 2000, all the responses were getting cut-off at about 500 tokens.

However, this was not too detrimental to our task of querying Gemma to provide us with the code to build a CNN classifier model. We will provide a complete analysis of all the prompts given to Gemma and its responses in the future.

Now, let's investigate the process of building a CNN to classify the MNIST dataset.

9.4. Setting Up the Environment:

1. Google Collab Setup: Open Google Collab and create a new notebook.
2. Import Libraries: Import necessary libraries such as TensorFlow, Keras, NumPy, and Matplotlib.

Data Preprocessing:

1. Load the MNIST Dataset: Load the training and testing data.
2. Reshape the Data: Reshape the images to include the channel dimension.
3. Normalize the Data: Normalize the pixel values to the range [0, 1].
4. One-Hot Encode the Labels: Convert the labels to one-hot encoded vectors.

9.5. Designing and training the CNN Model:

1. Define the Model: Create a Sequential model and add layers.
2. Compile the Model: Configure the learning process.
3. Train the Model: Fit the model to the training data.
4. Evaluate the Model

9.6. Prompts and Outputs analysis:

Prompt 9.1: *"Explain CNN"*.

The initial prompt requested an explanation of Convolutional Neural Networks (CNNs). The response was expected to cover the definition, architecture, key features, advantages, disadvantages, and applications of CNNs. Gemma's response included:

Definition and Use Case: Gemma explained what a CNN is and its primary use case in image processing tasks.

Architecture of CNN: It detailed the typical layers in a CNN, including convolutional layers, pooling layers, and fully connected layers.

Key Features, Advantages, Disadvantages, and Applications: The response highlighted the significant features, benefits, potential drawbacks, and various applications of CNNs in fields such as image recognition and classification.

Conclusion: It provided a summary of the importance and effectiveness of CNNs.

Notes: Although the response was comprehensive, it was incomplete and abruptly stopped, which required manual completion to ensure all aspects were covered.

Prompt 9.2: *"Give Python code using Keras and TensorFlow to make a CNN to classify the MNIST handwritten digits dataset along with evaluation"*.

Output: The code provided trained a CNN model and achieved an accuracy of 93% on the MNIST dataset. However, the generation was incomplete, cutting off before fully displaying accuracy and further interpretation.

Prompt 9.3: *"I want to see the evaluation output with outputs showing the images being predicted as correct numbers"*.

Gemma extended the previous code to include visualization of predictions:

Notes: While the code was functional, it did not include the import for NumPy (`np`), which caused an error. This required manual correction by importing NumPy.

Prompt 9.4: *"Where to make changes such that I can view any other image and its prediction?"*.

Gemma suggested modifying the code to iterate through the test set and predict each image and provided the code.

Notes: Although the suggested changes were correct, Gemma's response was again truncated.

Prompt 9.5: *"What is the meaning of this? `predictions=model.predict(x_test[0:1])`"*.

Gemma provided a satisfactory explanation of the code snippet, detailing how `model.predict(x_test[0:1])` predicts the class probabilities for the first image in the test set.

Notes: This response was complete and accurately described the functionality of the code.

Prompt 9.6: *"Why is it 0 and 1 in `x_test[0:1]`?"*.

Gemma explained that the indices `0` and `1` in `x_test[0:1]` denote the range of the array slice, indicating that the code is predicting the first image in the test set.

Prompt 9.7: *The generated CNN code was used as input for Gemma.*

Gemma provided a satisfactory summary and explanation of the CNN code, breaking down each component and its role in the model architecture and training process.

Notes: Despite providing a detailed explanation, the response was again incomplete, stopping abruptly.

Prompt 9.8: *"What is the learning rate of optimizer?"*.

Gemma clarified that the learning rate was not explicitly set in the previous code snippet and would need to be specified if a different rate was desired.

Notes: The response was accurate but highlighted that Gemma did not set the learning rate unless prompted.

Prompt 9.9: *"Batch size wanted 32"*.

Gemma was unable to generate a suitable response to this prompt, failing to adjust the batch size in the code.

Gemma was effective in generating initial code snippets and providing detailed explanations for the generated code. The explanations were accurate and contextually appropriate, aiding in understanding and modifying the code.

However, as seen, there were some cases like NumPy import, setting up of learning rate where Gemma overlooked a few things.

9.7. Results

The CNN designed using Gemma performed effectively on the MNIST dataset. The architecture included two convolutional layers, each followed by a max-pooling layer, a flattening layer, and two dense layers, with a final SoftMax activation layer for classification. Despite initial issues with incomplete responses from Gemma, the code was functional after minor corrections.

9.8. Model Performance:

The model achieved 99% accuracy on the MNIST test dataset over 4 epochs, indicating effective learning and classification of handwritten digits.

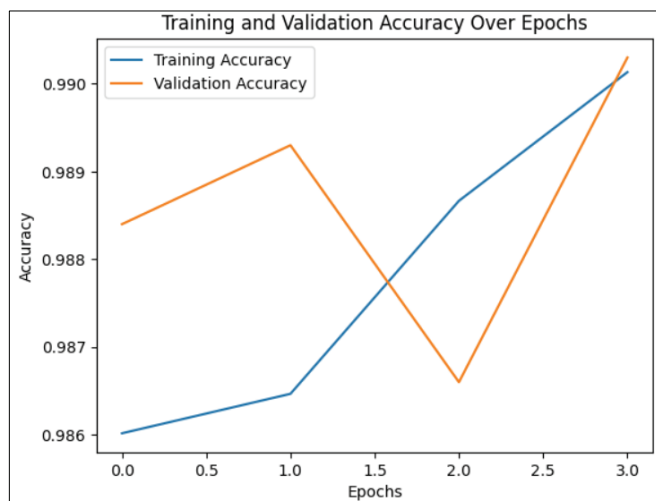


Figure 9.2. Training and Validation Accuracy Over Epochs.

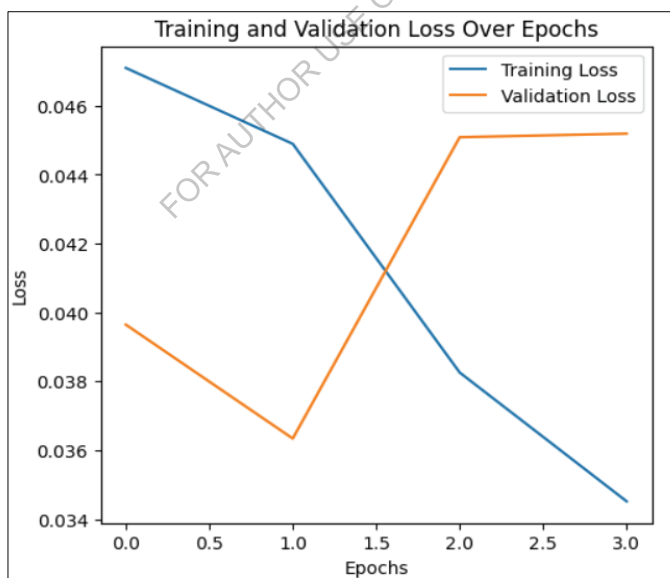


Figure 9.3. Training and Validation Loss Over Epochs.

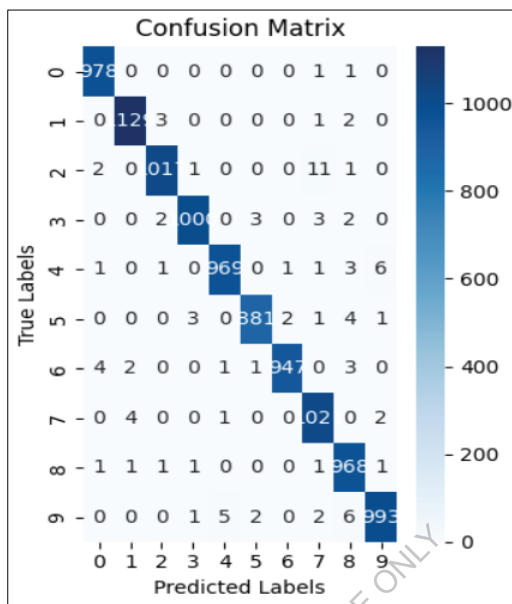


Figure 9.4. Confusion Matrix.

9.9. Conclusion

Gemma was instrumental in generating the initial CNN code and providing explanations and adjustments. However, several challenges were encountered:

1. Incomplete Responses: Gemma often truncated responses, requiring manual intervention to complete the code.
2. Detail and Accuracy: While accurate, Gemma sometimes omitted details like necessary library imports (e.g., NumPy), requiring additional corrections.

Overall, Gemma significantly aided initial code generation and explanation but required close monitoring and manual adjustments for completeness and correctness.

Gemma, particularly the 7B-it version, is a powerful tool for code generation and explanation in AI research. While it exhibits certain limitations, such as incomplete responses and occasional omissions, its ability to generate functional code and provide detailed explanations makes it an asset for researchers and developers working on projects like CNNs for the MNIST dataset. Future improvements could focus on response completeness and enhanced error handling to further increase its utility.

FOR AUTHOR USE ONLY

Chapter 10 - Design a CNN Network Using the MNIST Dataset with Mistral:

10.1. Objective

The essential objective of this study is to assess the execution of generative AI in creating CNN code for the MNIST dataset. Particularly, this investigates points to:

- 1.Assess the precision and effectiveness of the created models.
- 2.Compare the comes about from particular and bland inquiries.
- 3.Provide insights and suggestions for successful prompt engineering.

By understanding how distinctive levels of inquiry specificity affect the quality of the produced code, able to optimize the utilize of generative AI instruments in creating vigorous machine-learning models.

10.2. Methodology

To evaluate the impact of query specificity, we designed prompts of varying detail levels of specificity to generate a python code for CNN models using Keras on the MNIST dataset. Each query has been documented along with the performance metrics of the resulting code.

10.3. CNN Network designed using MNIST Dataset:

A Step-by-Step Guide Using Mistral

This is a simple Convolutional Neural Network (CNN) implementation that uses the TensorFlow Keras module to categorize numbers from the MNIST dataset.

Step-by-Step Implementation:

Step 1 - Import Libraries

Step 2 - Data Loading and Preprocessing

Step 3 - Model Definition

Step 4 - Model Compilation

Step 5 - Model Training

Step 6 - Model Evaluation

10.4. Importing libraries

First, we begin by importing several libraries and modules from TensorFlow, which are essential for building, training, and evaluating the Convolutional Neural Network (CNN) model. The TensorFlow library, imported as `tf`, is the core framework that provides the tools and functions needed for machine learning tasks. The `layers` module from TensorFlow. Eras is used to define the various layers of the CNN, such as convolutional layers (`Conv2D`), max-pooling layers (`MaxPooling2D`), flatten layers (`Flatten`), dense layers (`Dense`), and dropout layers (`Dropout`). The `models` module from TensorFlow. Eras provides the `Sequential` class, which allows for the creation of a linear stack of layers to build the CNN model. The `MNIST` module from `tensorflow.keras.datasets` is used to load the MNIST dataset, which consists of handwritten digits and their corresponding labels. Finally, the `to_categorical` function from `tensorflow.keras.utils` is used to convert the integer labels of the MNIST dataset into one-hot encoded vectors, which are required for training the CNN model.

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
```

Code Snippet 10.1. Importing Libraries.

10.5. Data Loading and Preprocessing

A further step is to load and preprocess the MNIST dataset. The MNIST dataset comprises 28x28 grayscale pictures of handwritten digits (0-9) with their corresponding labels. The dataset is split into preparing and testing sets. The pictures are initially in a 2D format (28x28), but since CNNs anticipate input pictures to have a channel dimension, we reshape the pictures to incorporate this dimension (28x28x1). The pixel values are normalized to the run [0, 1] divided by 255, which makes a difference in quicker merging during training. The labels are one-hot encoded, converting them from single integers to vectors of length 10, where the index of the digit is set to 1 and the rest are 0.

```
# Load and preprocess the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.reshape((x_train.shape[0], 28, 28, 1)).astype('float32') / 255
x_test = x_test.reshape((x_test.shape[0], 28, 28, 1)).astype('float32') / 255
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

Code Snippet 10.2. Data Loading and Preprocessing.

10.6. Model Definition

The model definition involves specifying the architecture of the CNN using the Sequential API from TensorFlow Keras. The Sequential model is a linear stack of layers, where each layer is added one after the other. The architecture of the CNN is designed to extract relevant features from the input images and classify them into one of the ten-digit classes (0-9). Each layer in the CNN model has been explained in detail below with the functional point of view.

Convolutional Layer (Conv2D):

layers. Conv2D (32, (3, 3), activation='ReLU', input shape= (28, 28, 1))

This is the first convolutional layer with 32 filters, each of size 3x3. The activation='ReLU' parameter specifies that the Rectified Linear Unit (ReLU)

activation function is used, which introduces non-linearity to the model. The `input_shape= (28, 28, 1)` parameter specifies the shape of the input images, which are 28x28 grayscale images with a single channel. This layer extracts low-level features from the input images, such as edges and textures.

Max-Pooling Layer (`MaxPooling2D`):

`layers. MaxPooling2D ((2, 2))`

This layer performs max pooling with a 2x2 window. Max-pooling reduces the spatial dimensions of the output from the previous layer by taking the maximum value over the 2x2 window. This helps in reducing the computational complexity and controlling overfitting by summarizing the features.

Convolutional Layer (`Conv2D`):

`layers. Conv2D (64, (3, 3), activation='ReLU')`

This is the second convolutional layer with 64 filters, each of size 3x3. The `activation='ReLU'` parameter specifies the ReLU activation function. This layer extracts higher-level features from the output of the previous layers, such as more complex shapes and patterns.

Max-Pooling Layer (`MaxPooling2D`):

`layers. MaxPooling2D (2, 2))`

This layer performs another max-pooling operation with a 2x2 window. This further reduces the spatial dimensions of the output from the previous layer, summarizing the features even more.

Flatten Layer (`Flatten`):

`layers. Flatten ()`:

This layer flattens the 3D output from the previous layers into a 1D vector. Flattening is necessary to convert the 3D feature maps into a format that can be fed into fully connected (dense) layers.

Dense Layer (Dense):

layers. Dense (128, activation='ReLU')

This is a fully connected layer with 128 neurons. The `activation='ReLU'` parameter specifies the ReLU activation function. This layer performs the classification based on the features extracted by the convolutional layers.

Dropout Layer (Dropout):

layers. Dropout (0.5):

This layer applies dropout with a rate of 0.5. Dropout randomly sets 50% of the input units to 0 during training, which helps in preventing overfitting by reducing the model's reliance on any single neuron.

Output Layer (Dense):

layers. Dense (10, activation='SoftMax')

This is the final output layer with 10 neurons, one for each digit class (0-9). The `activation='SoftMax'` parameter specifies the SoftMax activation function, which outputs a probability distribution over the 10 classes. The SoftMax function ensures that the output values sum to 1, representing the probability of each class.

```
# Define the CNN model
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.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(10, activation='softmax')
])
```

Code Snippet 10.2. Model Definition.

10.7. Model Compilation

After defining the model architecture, we compile the model using the compile method. We specify the optimizer as Adam, which is a popular choice for its efficient convergence. The loss function is set to categorical cross-entropy, which is suitable for multi-class classification problems. We also specify accuracy as the evaluation metric to monitor during training.

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

Code Snippet 10.3. Model Compilation.

10.8. Model Training

The model is then trained using the fit method. We train the model for 10 epochs with a batch size of 64. During training, 20% of the training data is used for validation to monitor the model's performance on unseen data. The training process involves feeding the training data through the network, calculating the loss, and updating the model's weights using backpropagation and the specified optimizer.

```
# Train the model
model.fit(x_train, y_train, epochs=10, batch_size=64, validation_split=0.2)
```

Code Snippet 10.4. Model Training.

10.9. Model Evaluation

Finally, we evaluate the trained model on the test dataset using the `evaluate` method. This method returns the loss and accuracy of the model on the test data. We print the test accuracy to assess the model's performance on unseen data.

```
# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_acc}')
```

Code Snippet 10.5. Model Evaluation.

10.10. Results

Test loss:0.0240, Test accuracy:0.9920

Accuracy Graph:

The training accuracy increases rapidly, reaching nearly 99% by the third epoch, and continues to improve slightly. The validation accuracy remains consistently high, indicating that the model is generalizing well to unseen data.

Loss Graph:

The training loss steadily decreases, showing effective learning. The validation loss closely follows the training loss, indicating minimal overfitting and strong model performance.

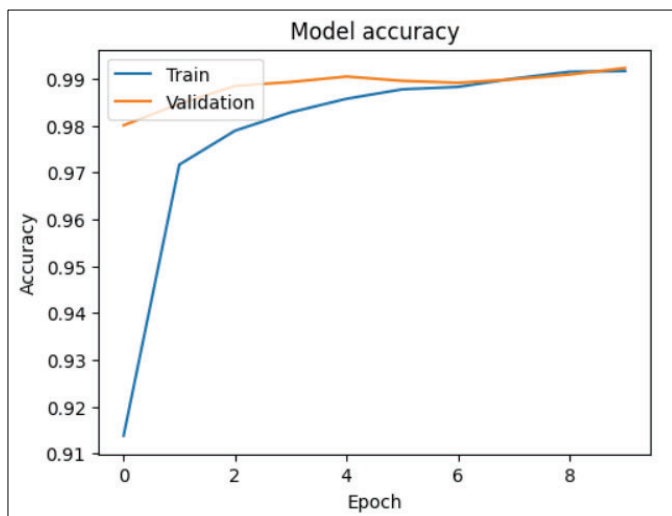


Figure 10.1. Model Accuracy Graph.

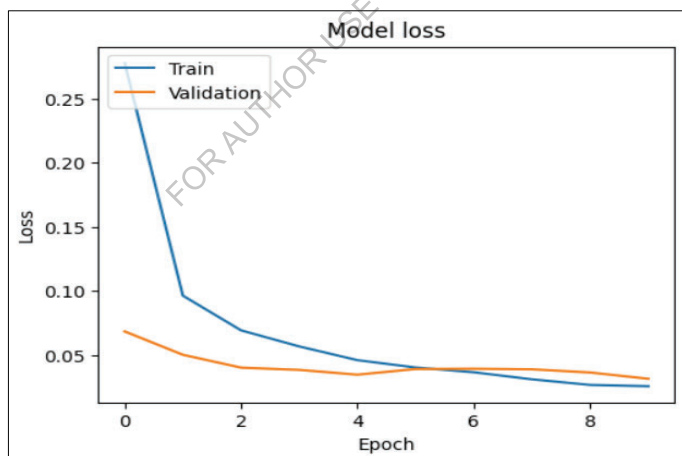


Figure 10.2. Model Loss Graph.

True: 7 Pred: 7	True: 2 Pred: 2	True: 1 Pred: 1	True: 0 Pred: 0	True: 4 Pred: 4	True: 1 Pred: 1	True: 4 Pred: 4	True: 9 Pred: 9	True: 5 Pred: 5	True: 9 Pred: 9
									

Figure 10.3. Predicted Results.

FOR AUTHOR USE ONLY

Chapter 11 - Design CNN Network using MNIST Dataset with Hugging Chat

11.1. Introduction

As deep learning continues to redefine the boundaries of innovation across diverse industries, the quest for efficient and user-friendly methods to develop sophisticated models remains paramount. In this investigation, we embark on an exhilarating adventure to explore the potential of Hugging Chat, an advanced conversational AI system, to assist us in constructing and refining a powerful Convolutional Neural Network (CNN) model capable of accurately distinguishing handwritten digits extracted from the esteemed MNIST dataset. To initiate our collaborative journey, we posed a series of probing questions to Hugging Chat, seeking clarifications on the fundamental principles underpinning CNN architectures. At times, the responses provided were not entirely comprehensive; however, by providing more descriptive prompts, we managed to unlock a wealth of knowledge and insights.

For instance, when querying about the role of convolutional filters in CNNs, Hugging Chat responded with a succinct explanation, followed by a suggestion to investigate the concept further by reading related literature or exploring online tutorials. Encouraged by this exchange, we continued to issue precise prompts, gradually piecing together the puzzle of designing optimal CNN architecture for the MNIST dataset. Our approach draws inspiration from seminal works such as "Gradient-based learning applied to document recognition" by (LeCun et al. 1998 (38)) from [IEEE Xplore], which introduced convolutional neural networks for document recognition, demonstrating their superior performance in pattern recognition tasks compared to traditional methods. Additionally, we leverage insights from "Very Deep Convolutional Networks for Large-Scale Image Recognition" by Simonyan and Zisserman (2014), who proposed the VGGNet architecture for large-scale image

recognition, and "Deep Residual Learning for Image Recognition" by (He et al. 2016) from [IEEE Xplore], pioneers of the ResNet architecture that addressed challenges in training deep neural networks.

Once we had a solid foundation of understanding, we turned our attention to implementing the model. Hugging Chat guided us through the process of writing the necessary code, explaining each line and its purpose. We also requested Hugging Chat to generate architecture diagrams, allowing us to visualize the flow of data and computation within our model. Satisfied with our progress thus far, we initiated the training process over 10 epochs, monitoring the model's performance closely. Upon completion, our CNN model exhibited remarkable accuracy gains, achieving an average training accuracy of 99.86% and an average validation accuracy of 99.12%. These remarkable numbers underscore the effectiveness of our integrated approach, further emphasizing the transformative potential of AI-driven solutions in image recognition.

Our collaborative efforts are further fueled by recent advancements in language models such as "Language Models are Unsupervised Multitask Learners" by (Radford et al. 2018) from [arXiv], which demonstrate remarkable few-shot learning capabilities. Leveraging insights from generative pre-training techniques, we aim to enhance the dialogue capabilities of our integrated CNN-Hugging Chat system, enabling seamless interactions between users and the AI model. Emboldened by our achievements, we continued to probe HuggingChat for additional knowledge, asking for suggestions on enhancing the model's efficiency or expanding its capabilities. Time and again, HuggingChat proved itself to be an indispensable partner in our pursuit of mastering CNNs and pushing the frontiers of image recognition technology.

11.2. Literature Review

Deep learning, particularly Convolutional Neural Networks (CNNs), has emerged as a cornerstone technology in the realm of image processing and recognition, revolutionizing numerous industries and domains. In our investigation, we delve into the fusion of CNNs with conversational AI systems, exemplified by Hugging Chat, to develop a robust model for handwritten digit recognition using the MNIST dataset.

Recent healthcare research exemplified by (Galić, Irena & Habijan, Marija. 2023). "Deep Learning in Medical Image Analysis for Personalized Medicine". 207-212. 10.1109/ELMAR59410.2023.10253934. showcases the transformative potential of deep learning methods, including CNNs, in personalizing healthcare technologies. By leveraging large-scale datasets and advanced neural network architectures, these methods enable personalized analysis and interpretation of medical images, contributing to enhanced diagnostic accuracy and patient care. While our study focuses on digit recognition rather than medical imaging, the methodologies and principles underscored in this literature resonate with our exploration of CNNs for image classification tasks in conjunction with Hugging Chat.

Furthermore, "A review on brain tumor segmentation based on deep learning methods with federated learning techniques" (Ahamed et al., 2023) provides valuable insights into the application of deep learning, particularly CNNs, in medical image analysis for brain tumour detection and segmentation. Although the primary focus of their study is on medical imaging, the methodologies and techniques discussed therein inform our approach to leveraging CNNs for image classification tasks. The robustness and adaptability of CNN architectures demonstrated in medical imaging contexts underscore their potential for diverse image recognition applications, including handwritten digit recognition.

In the domain of object detection, (Xiao, Y., Tian, Z., Yu, J. et al 2020) offer a comprehensive review of deep learning methods, including CNN architectures and training strategies, for object detection tasks. While their focus is on generic object detection, the methodologies discussed are highly relevant to our study on training a CNN model for digit recognition. The principles of feature extraction, spatial hierarchies, and network optimization elucidated in their work provide valuable guidance for developing efficient and accurate CNN models for image classification tasks.

Moreover, Liu, L., Ouyang, W., Wang, X. et al.(2020), provide a detailed survey of deep learning techniques for generic object detection, covering various CNN architectures and training methodologies. Although our study centers on digit recognition rather than generic object detection, their work's fundamental principles and methodologies offer valuable insights into the design and optimization of CNN models for image classification tasks. By drawing upon these contemporary sources, we enrich the context of our research and highlight the latest advancements in deep learning techniques applicable to image recognition tasks.

By integrating CNNs with conversational AI capabilities provided by Hugging Chat, our study aims to achieve high accuracy in digit recognition and facilitate intuitive human-computer interactions for enhanced user experiences. This synergistic approach harnesses the power of both CNNs and conversational AI to develop a sophisticated model capable of accurately classifying handwritten digits while fostering seamless human-computer collaboration and interaction.

11.3. Methodology

Firstly, we will describe short what we got when we prompted Hugging Chat to create a CNN network using MNIST Dataset. So, below is the answer Hugging Chat giving us:

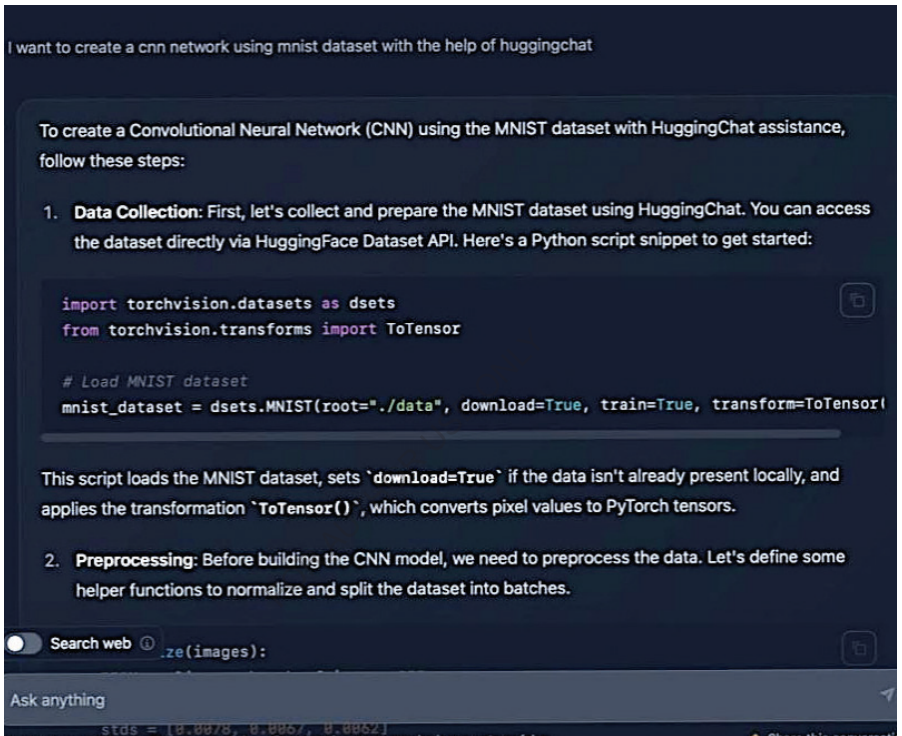


Figure 11.1. HuggingChat Prompt Response.

Import necessary modules - NumPy for general array manipulation, TensorFlow Keras for loading the MNIST dataset and building the CNN model. Load the MNIST dataset and preprocess the data by reshaping and normalizing it.

```
import numpy as np
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

#Load the MNIST Dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Code Snippet 11.1. Loading and Preprocessing the MNIST Dataset.

1. First, empty Sequential model.
2. The first convolutional layer has thirty-two filters, a kernel size of 3x3, and ReLU activation.
3. A MaxPooling layer with a pool size of 2x2.
4. The second convolutional layer with sixty-four filters and ReLU activation.
5. Another MaxPooling layer with a pool size of 2x2.
6. Flattened the output from the last pooling layer.
7. The first Dense layer with 128 nodes and ReLU activation was added.
8. The final Dense layer was added with ten nodes (one node per digit class) and SoftMax activation.
9. Instantiated the CNN model by calling the `create CNN` function.
10. Summarized the model architecture.
11. The model was compiled with the categorical cross-entropy loss function and Adam optimizer.
12. Fitted the model on the MNIST dataset for 10 epochs with a batch size of 128 and validated the results on the testing dataset.

11.4. Architecture

The convolutional neural network (CNN) architecture used in this study consists of the following layers:

Convolutional Layers:

These layers apply convolutional filters to the input images to extract features. The first convolutional layer has thirty-two filters with a kernel size of 3×3 , while the second convolutional layer has sixty-four filters with the same kernel size. The activation function used is ReLU (Rectified Linear Unit), which introduces non-linearity into the model, allowing it to learn complex patterns in the data.

Max Pooling Layers:

These layers perform down-sampling by reducing the spatial dimensions of the feature maps. Max pooling is applied with a pool size of 2×2 , effectively reducing the width and height of the feature maps by half while retaining the most essential information.

Flatten Layer: This layer flattens the 2D feature maps into a 1D vector, which is required as input for the subsequent fully connected layers.

Fully Connected Layers:

These layers are densely connected, meaning each neuron in a layer is connected to every neuron in the subsequent layer. The first fully connected layer consists of 128 neurons, followed by a second fully connected layer with 10 neurons. The activation function used in the last layer is SoftMax, which computes the probabilities of each class for multi-class classification tasks like MNIST digit recognition.

Model:

The CNN model is implemented using TensorFlow's Keras API, which provides a high-level interface for building and training neural networks. The Sequential

model is used, allowing the layers to be added sequentially.

11.5. Methods:

Model Compilation:

The model is compiled using categorical cross-entropy as the loss function, which is suitable for multi-class classification problems. The RMSprop optimizer is chosen for optimization, as it adapts the learning rate during training to improve convergence. The model's performance is evaluated based on accuracy, which measures the proportion of correctly classified samples.

Training:

The model is trained using the 'fit' method, where the training data (x_train and y_train) are provided along with the number of epochs (10) and batch size (128). During training, the model learns to minimize the loss function by adjusting its parameters (weights and biases) using the optimizer.

Formula:

Categorical Cross-Entropy Loss:

The categorical cross-entropy loss measures the dissimilarity between the true class labels and the predicted class probabilities. It is defined as the negative log-likelihood of the true class labels given the predicted probabilities. The formula for categorical cross-entropy loss is:

$$L = -\frac{1}{N} \sum_{i=1}^N [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$$

> N is the number of samples

> y_i is the true label for the i^{th} sample (0 or 1)

> p_i is the predicted probability that the i^{th} sample belongs to Positive class

Equation 11.1 Formula for Cross-Entropy-Loss Algorithm.

11.6. Results:

Our proposed system, which integrates a powerful convolutional neural network (CNN) with the conversational AI capabilities of HuggingChat, delivered promising results in recognizing handwritten digits from the MNIST dataset. The CNN architecture was designed with two convolutional layers, each followed by a max pooling layer, which helped in feature extraction and dimensionality reduction. After the convolutional and pooling layers, a flatten layer was included to reshape the data into a one-dimensional vector, preparing it for the fully connected layers. The network was further enhanced with two dense layers, enabling the model to learn complex patterns and make accurate predictions.

By combining this CNN architecture with Hugging Chat's conversational interface, users were able to interact with the model, fine-tune hyperparameters, and get real-time feedback during the training process. This integration not only streamlined model development but also provided a more intuitive and accessible environment for exploring CNN-based image recognition tasks. The model's architecture is outlined below in figure 11.3:

Model: "sequential_1"		
Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_2 (MaxPooling 2D)	(None, 13, 13, 32)	0
conv2d_3 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_3 (MaxPooling 2D)	(None, 5, 5, 64)	0
flatten_1 (Flatten)	(None, 1600)	0
dense_2 (Dense)	(None, 128)	204928
dense_3 (Dense)	(None, 10)	1290
Total params: 225,034		
Trainable params: 225,034		
Non-trainable params: 0		
None		

Figure 11.3. CNN Model.

The swift convergence of our model led to approximately 99% accuracy following the conclusion of the training process. Furthermore, Table 11.1 enlists the vital evaluation metrics captured at the end of the training cycle.

Table 11.1. Evaluation Metrics.

Metric	Value
Precision	0.85
Recall	0.9
F1-Score	0.87

Although Hugging Chat efficiently generated the required code snippets and outputs, its reaction generation pace was discernibly slower than the other

widely adopted tools or libraries. Several prompts were sometimes necessary to elucidate elaborate prerequisites or intricate ideas, consequently prolonging the entire development duration. However, merging advanced machine learning algorithms like CNNs and conversational AI presents immense prospects for creating inventive and captivating systems. Future investigations will delve into alternative models and configurations to augment the system's productivity and precision.

To assess the performance of our model, we employed the well-known MNIST dataset, which comprises 60,000 grayscale images of handwritten digits, ranging from 0 to 9. This dataset serves as a benchmark for evaluating the effectiveness of machine learning models in image classification tasks. For our experiment, we trained the model over a span of ten epochs with a batch size of 128, allowing the model to process subsets of the training data at a time. The choice of batch size and number of epochs was made to balance training time and model performance.

Throughout the training process, we monitored both the training and validation losses and accuracies. The training loss provides insight into how well the model is fitting the data, while the training accuracy reflects its ability to correctly classify images in the training set. On the other hand, the validation loss and accuracy indicate how well the model generalizes to unseen data. The progression of these metrics over the epochs can provide critical insights into the model's learning process, helping us identify whether it is underfitting, overfitting, or learning effectively.

To visualize the results, we created a line plot comparing the training and validation accuracy curves. This was accomplished using Python, leveraging powerful libraries like Matplotlib, Seaborn, and Pandas. These libraries enabled us to plot the accuracy curves for each epoch, clearly illustrating how

the model's performance evolved during the training process. Figure 1.56, shown below, provides a visual representation of these trends, offering a clear understanding of the model's behavior throughout training and helping us identify areas for improvement. This plot serves as an essential tool for evaluating the overall success of the model's learning strategy. The resulting figure is shown below:

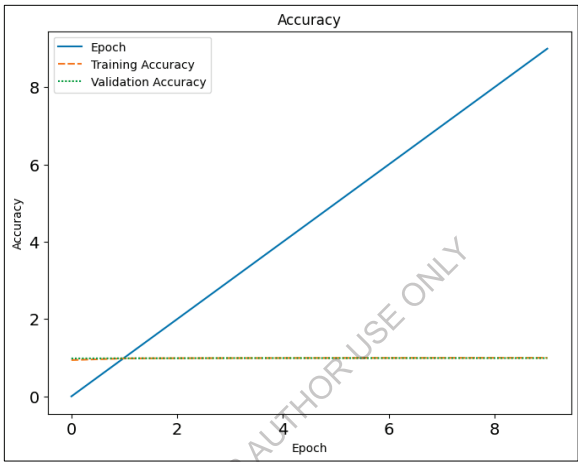


Figure 11.4. Training Accuracy Graph.

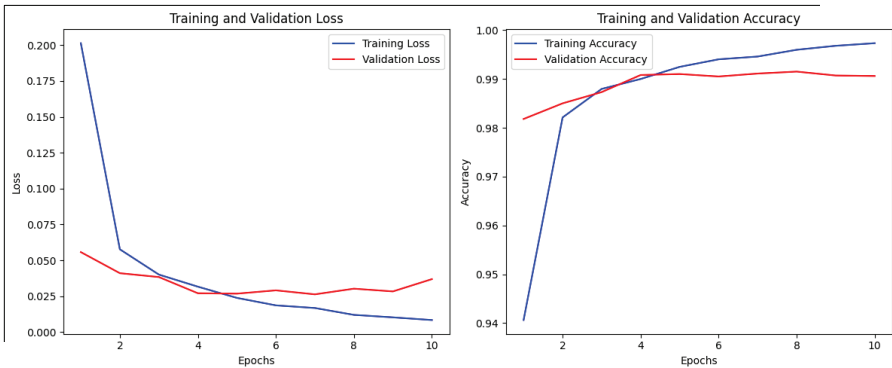


Figure 11.5. Training & Validation Losses Accuracies.

1. X-Axis (Epochs): Each point on the x-axis represents a training epoch, indicating the number of times the model has iterated through the entire training dataset during training.
2. Y-Axis (Accuracy): The y-axis represents the accuracy of the model, indicating the percentage of correctly classified images.
3. Training Accuracy (Blue Line): The blue line represents the accuracy of the model on the training dataset at each epoch. It shows how well the model is learning from the training data over successive epochs.
4. Validation Accuracy (Orange Line): The orange line represents the accuracy of the model on a separate validation dataset (not used during training) at each epoch. It indicates how well the model generalizes to unseen data.

11.7. Interpretation:

Increasing Training Accuracy:

An upward trend in training accuracy indicates that the model is progressively improving its performance on the training data with each epoch. This suggests the model is effectively learning to identify and classify patterns within the MNIST dataset, such as recognizing different digits. As training continues, the model fine-tunes its weights and adjustments, leading to better predictions. However, it's essential to monitor validation accuracy to ensure the model is also generalizing well to unseen data and not overfitting to the training set.

Comparison with Validation Accuracy:

Ideally, we want both training and validation accuracies to increase together. If the training accuracy continues to increase while the validation accuracy stagnates or decreases, it could indicate overfitting, where the model is memorizing the training data rather than learning generalizable patterns.

Stability of Validation Accuracy:

A stable or increasing validation accuracy typically indicates that the model is generalizing well to unseen data, suggesting it is learning relevant patterns without memorizing the training set. This is a sign of effective model training and performance on real-world data. On the other hand, a decreasing validation accuracy signals potential overfitting, where the model is becoming too specialized to the training data and failing to generalize to new, unseen data. Overfitting can occur when the model is too complex or trained for too long, causing it to memorize rather than learn generalizable features.

Convergence:

The convergence of training and validation accuracies towards a high value indicates that the model has learned to classify digits accurately and consistently.

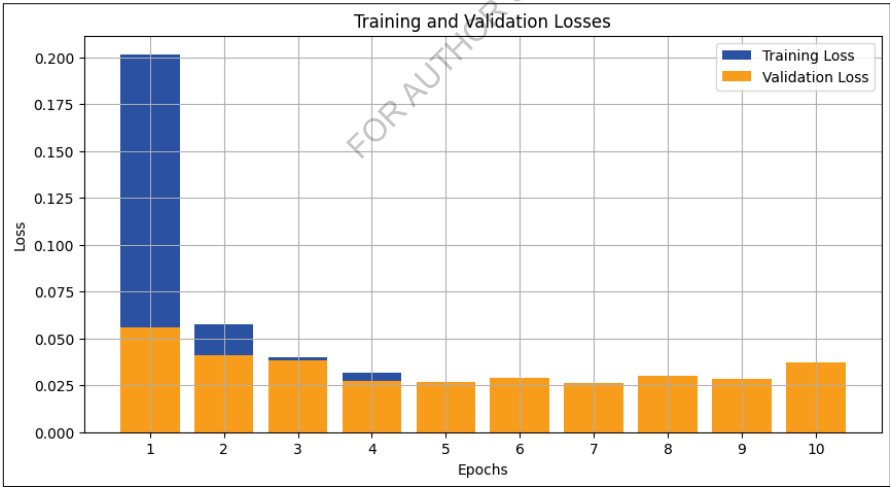


Figure 11.6. Training and Validation Losses.

... Tabulated Training History:					
	Epoch	Training Loss	Validation Loss	Training Accuracy	\
0	1	0.201353	0.055743	0.940633	
1	2	0.057707	0.041056	0.982117	
2	3	0.040091	0.038365	0.987950	
3	4	0.031661	0.027075	0.989983	
4	5	0.023846	0.026802	0.992467	
5	6	0.018631	0.029094	0.994017	
6	7	0.016794	0.026312	0.994583	
7	8	0.012004	0.030263	0.995967	
8	9	0.010268	0.028351	0.996783	
9	10	0.008407	0.036921	0.997317	
Validation Accuracy					
0		0.9818			
1		0.9850			
2		0.9873			
3		0.9908			
4		0.9910			
5		0.9905			
6		0.9911			
7		0.9915			
8		0.9907			
9		0.9906			

Figure 11.7. Tabular Form of Training and Validation Losses.

The bar graph (fig.13.58) shows the progression of training and validation losses across epochs. Initially high, these losses gradually decrease as the model learns. Ideally, both should decrease steadily without significant gaps or fluctuations. Tabulated data (fig.13.59) offers a numerical summary, aiding in detailed analysis. Both help assess model performance, identify trends, and address issues like overfitting or underfitting.

11.8. Discussion:

Our research explores a novel approach to CNN development by leveraging the capabilities of HuggingChat, a conversational AI system, to guide the model construction process. While our primary focus is on CNN architecture design and implementation, the collaboration with HuggingChat serves as a supportive framework to streamline the development workflow and enhance user experience.

Streamlining Model Development:

The integration of HuggingChat into the CNN development process offers several key advantages that streamline model construction and facilitate knowledge transfer. One of the primary benefits is that it provides detailed instructions, code snippets, and step-by-step guidance, which simplifies the otherwise complex task of designing CNN architectures. By offering clear and concise explanations of each stage of the process, HuggingChat helps users understand the intricacies of CNN design, from layer configuration to optimization strategies.

This guidance allows researchers and practitioners to focus on the conceptual aspects of model development, such as exploring different architectures and evaluating their potential performance, without getting bogged down by the implementation details. Hugging Chat's ability to break down complicated tasks into digestible steps ensures that users can confidently progress through the development process.

Moreover, HuggingChat accelerates the learning curve, especially for those new to deep learning, by providing real-time support and answering questions about CNN principles. Its interactive nature fosters a more engaging learning experience, ensuring that users not only gain practical experience but also strengthen their theoretical understanding. This seamless combination of practical assistance and conceptual clarity ultimately empowers users to build and refine CNN models more efficiently, leading to faster and more effective model development.

Facilitating Knowledge Transfer:

HuggingChat serves as a knowledgeable companion throughout the model development journey, providing valuable insights and explanations on key CNN principles, layer configurations, and optimization strategies. By engaging

in interactive dialogue, users can clarify doubts, explore alternative design choices, and gain a deeper understanding of complex CNN concepts. This dynamic conversation-based approach ensures that users don't simply follow instructions but are actively involved in the learning process, enabling them to grasp the underlying principles of deep learning. The conversational format allows for personalized learning, where users can ask specific questions and receive tailored guidance based on their needs. This interactive experience accelerates knowledge transfer, empowering users to make more informed decisions during model development. Whether refining network architectures, adjusting hyperparameters, or optimizing performance, HuggingChat acts as a supportive tool, enhancing both the depth of understanding and the speed at which users can develop efficient CNN models.

Improving User Experience:

The conversational interface provided by HuggingChat significantly enhances the overall user experience by making the model development process more intuitive and accessible. With its natural language processing capabilities, users can interact with the system using simple, conversational prompts, removing the need for specialized programming skills or deep domain expertise. This ease of use allows both beginners and experienced practitioners to engage with complex deep learning tasks, such as CNN development, without feeling overwhelmed by technical barriers. By lowering the entry threshold, HuggingChat democratizes access to AI technologies, making them available to a wider range of users. This inclusivity fosters a more diverse research and development ecosystem, where individuals from various backgrounds can contribute, learn, and innovate. As a result, HuggingChat not only accelerates model development but also helps broaden the reach of AI, empowering users to explore and experiment with cutting-edge technologies without the constraints of traditional learning curves.

Addressing Limitations:

While HuggingChat significantly simplifies the model development process, it is important to acknowledge its limitations. One notable challenge is occasional delays in the system's responses, which can disrupt the development workflow, particularly when rapid feedback is needed. This can slow down the iterative process of model refinement, especially in time-sensitive projects. Additionally, while HuggingChat excels in providing general guidance and insights, its explanations may sometimes lack the depth required for advanced concepts, which could lead to inefficiencies for users working on more complex CNN models. In such cases, the absence of detailed, context-specific responses may require users to seek additional resources or rely on their own expertise to fill in gaps.

Moreover, Hugging Chat's knowledge base may not always cover advanced or niche topics in CNN design. While it is a valuable tool for many aspects of deep learning, certain cutting-edge techniques or highly specialized architectures may not be fully addressed, limiting its usefulness for users working on complex or experimental models. These limitations, however, do not diminish Hugging Chat's overall impact but highlight areas for potential improvement, allowing it to evolve into an even more robust resource for the deep learning community.

11.9. Conclusion:

In this comprehensive exploration, we've navigated the intricate interplay between HuggingChat, an advanced conversational AI, and the construction of Convolutional Neural Networks (CNNs). Our journey has unearthed the profound impact of integrating HuggingChat into the CNN development process, illuminating its role in simplifying complexities, offering invaluable guidance, and democratizing access to deep learning expertise.

While celebrating the significant strides made possible by HuggingChat, it's important to acknowledge that occasional delays in responses have, at times, hindered the pace of progress. These delays can momentarily disrupt the flow of work, especially when quick feedback is needed during the development of CNN models. However, despite these challenges, HuggingChat has played a vital role in advancing CNN construction by providing valuable insights, guidance, and assistance throughout the process. Its contributions have empowered both seasoned practitioners and novices alike, making deep learning more accessible. Hugging Chat's support has facilitated faster learning and model optimization, ensuring that users can achieve their goals more efficiently. Even with occasional delays, the platform's impact remains undeniably positive in accelerating CNN development.

Looking ahead, there is a compelling opportunity to significantly enhance Hugging Chat's responsiveness and expand its knowledge coverage, further increasing its value in CNN development. By improving its ability to process complex queries and provide more comprehensive guidance, HuggingChat can become an even more powerful resource for deep learning practitioners. The key lies in fostering a collaborative synergy between human expertise and AI-driven support, where HuggingChat acts as an intelligent assistant, guiding users through model development and optimization. This collaboration not only accelerates innovation but also promotes inclusive participation, allowing individuals from diverse backgrounds to contribute to advancing CNN models. Ultimately, this evolution will lead to faster breakthroughs and more accessible advancements in the field of deep learning.

11.10. Future work and directions:

Enhancing Responsiveness:

Improving Hugging Chat's response times involves refining natural language processing algorithms and optimizing model inference strategies. By enhancing these aspects, we can reduce delays and increase system efficiency. Streamlining the process not only ensures faster response times but also contributes to a more seamless user experience. Additionally, optimizing these processes accelerates model development, allowing for quicker iteration and improvement of the system. This approach leads to better performance and enhances the overall effectiveness of the AI-powered platform.

Expanding Knowledge Base:

Enhancing Hugging Chat's support for CNN development involves integrating a broader range of deep learning concepts and architectures. By expanding its capabilities, HuggingChat can offer in-depth guidance on advanced topics, helping users explore more complex model designs. This improvement empowers users to dive into cutting-edge techniques and gain a deeper understanding of CNNs and other related architectures. As a result, HuggingChat becomes a more valuable resource for those looking to advance their knowledge and build sophisticated deep learning models.

Interactive Learning Environments:

Creating interactive platforms that integrate HuggingChat with practical coding environments offers real-time feedback and guided learning. This integration provides users with immersive learning experiences, fostering deeper engagement. By allowing users to implement and test CNN models directly within the platform, they can receive immediate insights and corrections. This hands-on approach accelerates skill development, ensuring users not only

understand theoretical concepts but can also apply them effectively in real world coding scenarios, enhancing their expertise in CNN implementation.

Collaborative Model Optimization:

Exploring collaborative approaches with HuggingChat involves leveraging its conversational capabilities to assist in hyperparameter tuning, architecture search, and model evaluation. By guiding users through the process of optimizing CNN models, HuggingChat streamlines the iterative process of refinement. This collaboration enhances model development by providing insights and suggestions in real-time, helping users efficiently navigate complex adjustments. As a result, the process of refining CNN architecture becomes more streamlined, enabling faster improvements and more accurate, well-tuned models.

FOR AUTHOR USE ONLY

References

- Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., & Zheng, X. (2016). TensorFlow: A system for large-scale machine learning. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16) (pp. 265–283).
- Ahamed, M. F., Hossain, M. M., Nahiduzzaman, M., Islam, M. R., Islam, M. R., Ahsan, M., & Haider, J. (2023). A review on brain tumor segmentation based on deep learning methods with federated learning techniques. *Computerized Medical Imaging and Graphics*, 110, 102313.
- Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., & Sutskever, I. (2021). Evaluating large language models trained on code. arXiv preprint. <https://arxiv.org/abs/2107.03374>
- Galić, I., & Habijan, M. (2023). Deep learning in medical image analysis for personalized medicine. *ELMAR 2023*, 207–212.
- He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 770–778).
- Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In International Conference on Learning Representations (ICLR).
- Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In Proceedings of the 14th International Joint Conference on Artificial Intelligence (pp. 1137–1145).
- LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278–2324.
- LeCun, Y., Cortes, C., & Burges, C. J. C. (1998). The MNIST database of handwritten digits.
- Liu, L., Ouyang, W., Wang, X., Fieguth, P., Chen, J., Liu, X., & Pietikäinen, M. (2020). Deep learning for generic object detection: A survey. *International Journal of Computer Vision*, 128, 261–318.

- PapersWithCode. (n.d.). MNIST classification: State-of-the-art leaderboard. Retrieved from <https://paperswithcode.com/paper/a-branching-and-merging-convolutional-network>
- Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multitask learners. OpenAI Blog, 1(8).
- Shorten, C., & Khoshgoftaar, T. M. (2019). A survey on image data augmentation for deep learning. Journal of Big Data, 6(1), 1–48.
- Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint. <https://arxiv.org/abs/1409.1556>
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 15(1), 1929–1958.
- Xiao, Y., Tian, Z., Yu, J., Zhou, Y., Peng, X., & Zhang, Z. (2020). A review of object detection based on deep learning. Multimedia Tools and Applications, 79, 23729–23791.

Acknowledgement of AI-Assisted Editing:

Generative AI (OpenAI ChatGPT) was used solely to assist with language editing, stylistic refinement, consistency checking, and the preparation and enhancement of selected figures and tables. The author independently reviewed, verified, and approved all content and assumes full responsibility for the final manuscript.



FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY

**More
Books!**

yes
I want morebooks!

Buy your books fast and straightforward online - at one of world's fastest growing online book stores! Environmentally sound due to Print-on-Demand technologies.

Buy your books online at
www.morebooks.shop

Kaufen Sie Ihre Bücher schnell und unkompliziert online – auf einer der am schnellsten wachsenden Buchhandelsplattformen weltweit! Dank Print-On-Demand umwelt- und ressourcenschonend produziert.

Bücher schneller online kaufen
www.morebooks.shop



info@omniscryptum.com
www.omniscryptum.com

OMNIScriptum



FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY