# Task 2: CNN Model on the MNIST Dataset

By Ramanand R

# Table of Contents

1. Introduction

2. Installation and Environment Setup

* Framework Installation (TensorFlow/Keras)
* Setting up the Development Environment

3. Dataset Preparation

* Loading the MNIST Dataset
* Data Splitting and Preprocessing

4. Model Architecture and Training

* Custom CNN Model
* Model Design
* Model Compilation
* Model Training
* Transfer Learning with VGG19
* Model Setup (VGG19)
* Model Compilation
* Model Training

5. Model Evaluation

* Evaluation Metrics
* Accuracy
* Precision
* Recall
* F1-Score
* Confusion Matrices
* Visualization of Performance Metrics

6. Overfit Analysis

* Monitoring Training and Validation Curves
* Mitigating Overfitting (if necessary)

7. Conclusion

# 1. Introduction

##### Installation and Environment Setup

In this section, we will walk through the installation process of the essential frameworks, TensorFlow and Keras, and set up the development environment to prepare for training and evaluating our convolutional neural network (CNN) models on the MNIST dataset.

**Framework Installation (TensorFlow/Keras)**

TensorFlow and Keras are fundamental libraries for deep learning tasks, and we need to ensure they are properly installed in our environment. Below are the steps to install them:

*Installing TensorFlow*

1. Open your command prompt or terminal.

2. Depending on your system, you can install TensorFlow for CPU or GPU. For most users, installing the CPU version is sufficient for this task.

To install the CPU version, use the following command:

pip install tensorflow

To install the GPU version (if you have a compatible GPU), use the following command:

pip install tensorflow-gpu

3. TensorFlow will be downloaded and installed automatically. Once the installation is complete, you can verify it by importing TensorFlow in a Python script or an interactive Python environment.

*Installing Keras*

1. Keras is now integrated with TensorFlow as its high-level API. Therefore, once TensorFlow is installed, Keras is also available.

2. You can test Keras by importing it in your Python script or environment:

import tensorflow as tf

from tensorflow import keras

Now that we have TensorFlow and Keras installed, let's move on to setting up the development environment.

*Setting up the Development Environment*

To create, train, and evaluate CNN models effectively, it's essential to have a well-configured development environment. Here are the key steps:

1. *Python Environment*: Ensure you have Python installed (preferably Python 3.6 or higher) on your system.

2. *Virtual Environment (Optional)*: Consider creating a virtual environment for your project to manage dependencies. You can create one using `virtualenv` or `conda`.

3. *IDE or Text Edito*r: Choose an Integrated Development Environment (IDE) or text editor of your preference. Popular choices include Visual Studio Code, PyCharm, Jupyter Notebook, or even a text editor like Sublime Text.

1. *Package Management*: Use `pip` or `conda` (if using Anaconda) to manage Python packages and dependencies.

5. *Data Storage*: Organize your project directory and determine where you'll store datasets and model files.

6. *Jupyter Notebook (Optional)*: Consider using Jupyter Notebook for interactive development and experimentation. Install Jupyter by running `pip install jupyter`.

7. *GPU Support (Optional):* If you have a compatible GPU and installed the GPU version of TensorFlow, you can leverage GPU acceleration for faster training.

With TensorFlow, Keras, and your development environment in place, we're ready to proceed with loading the MNIST dataset and training our CNN models.

# 2. Installation and Environment Setup

To successfully train and evaluate convolutional neural network (CNN) models on the MNIST dataset, it's crucial to have the necessary libraries installed and a well-configured development environment. In this section, we will explain the steps to install the required libraries, TensorFlow and Keras, and set up your development environment.

*2.1. Framework Installation (TensorFlow/Keras)*

2.1.1. Installing TensorFlow

TensorFlow is an open-source machine learning framework developed by Google. It serves as the backend for Keras, which is a high-level neural networks API. Follow these steps to install TensorFlow:

1. *Open a Command Prompt or Terminal* : Depending on your operating system, open a command prompt or terminal window.

2. *Install TensorFlow*: Use `pip` to install TensorFlow. If you have a CPU-only machine, you can install the CPU version. If you have a compatible NVIDIA GPU, you can install the GPU version for faster training:

For CPU version:

pip install tensorflow

For GPU version (requires NVIDIA GPU and CUDA support):

pip install tensorflow-gpu

3. Verification: After installation, you can verify if TensorFlow is installed correctly by running a Python script and importing TensorFlow:

import tensorflow as tf

2.1.2. Installing Keras

Keras is a high-level neural networks API that runs on top of TensorFlow (or other deep learning frameworks). Since TensorFlow 2.0, Keras is included within TensorFlow. You can import Keras modules directly after installing TensorFlow:

from tensorflow import keras

2.2. Setting up the Development Environment

To create an environment for developing and training CNN models on the MNIST dataset, follow these steps:

1. Python Environment: Ensure you have Python installed on your system. TensorFlow and Keras are compatible with Python 3.6 or higher.

2. Virtual Environment (Optional\*: Consider creating a virtual environment for your project to isolate dependencies. You can create one using tools like `virtualenv` or `conda`:

Using `virtualenv`:

pip install virtualenv

virtualenv myenv

source myenv/bin/activate # On Windows, use `myenv\Scripts\activate`

Using `conda` (if Anaconda is installed):

conda create --name myenv python=3.8

conda activate myenv

3. IDE or Text Editor: Choose an Integrated Development Environment (IDE) or text editor based on your preference. Popular options include Visual Studio Code, PyCharm, Jupyter Notebook, or plain text editors like Sublime Text or VS Code.

4. Package Management: Use `pip` or `conda` to manage Python packages and dependencies within your virtual environment. You will need to install additional packages for data manipulation, visualization, and evaluation as you progress with your project.

5. Project Structure: Organize your project directory. Consider creating subdirectories for datasets, trained models, and source code. This helps maintain a clean and structured project.

6. Jupyter Notebook (Optional): If you prefer interactive development and experimentation, consider using Jupyter Notebook. Install Jupyter by running:

pip install jupyter

7. GPU Support (Optional): If you have a compatible NVIDIA GPU and installed the GPU version of TensorFlow, you can take advantage of GPU acceleration for faster model training. Ensure your GPU drivers and CUDA are properly configured.

With TensorFlow, Keras, and your development environment set up, you're ready to proceed with loading the MNIST dataset and training CNN models for digit classification.

## **3. Dataset Preparation**

In this section, we'll dive into the process of preparing the MNIST dataset for training and evaluation of our convolutional neural network (CNN) models. We'll explain how the dataset is loaded, how it's split into training and testing sets, and any preprocessing steps applied to the data.

*3.1. Loading the MNIST Dataset*

The MNIST dataset is a widely used dataset in the field of machine learning and computer vision. It consists of a large number of grayscale images of handwritten digits (0 to 9), each of which is 28 pixels in height and 28 pixels in width.

*Loading MNIST Using TensorFlow/Keras:*

We'll load the MNIST dataset using the TensorFlow/Keras library, which provides a convenient way to access and work with the dataset. Here are the steps to load the dataset:

Code:

from tensorflow.keras.datasets import mnist

# Load the MNIST dataset

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

- `x\_train` and `y\_train` contain the training images and corresponding labels, respectively.

- `x\_test` and `y\_test` contain the testing images and labels.

*3.2. Splitting the Dataset*

To train and evaluate our CNN models, we need to split the dataset into training and testing sets. A common split ratio is 80% for training and 20% for testing. We can also choose to set aside a portion of the training data for validation to monitor the model's performance during training.

*Splitting Using scikit-learn (Optional):*

If you prefer to use scikit-learn to split the dataset, you can follow these steps:

from sklearn.model\_selection import train\_test\_split

# Split the dataset into training and testing sets

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x\_train, y\_train, test\_size=0.2, random\_state=42)

# Optionally, create a validation set

x\_train, x\_val, y\_train, y\_val = train\_test\_split(x\_train, y\_train, test\_size=0.2, random\_state=42)

- `x\_train` and `y\_train` now contain the training images and labels.

- `x\_test` and `y\_test` contain the testing images and labels.

- `x\_val` and `y\_val` (optional) contain the validation images and labels.

*3.3. Preprocessing*

**Normalization**

Before feeding the images into our CNN models, it's essential to preprocess them. The most common preprocessing step is normalization, which scales the pixel values to be in the range [0, 1]. This ensures that the neural network learns more effectively and converges faster.

# Normalize pixel values to be between 0 and 1

x\_train, x\_val, x\_test = x\_train / 255.0, x\_val / 255.0, x\_test / 255.0

- We divide all pixel values by 255.0 to rescale them to the [0, 1] range.

With the dataset loaded, split, and preprocessed, we're ready to design and train our CNN models for digit classification. The next section will cover the model architecture and training process.

# 4. Model Architecture and Training

In this section, we will delve into the architecture, compilation, and training processes of the two models used in this project: the custom CNN model and the Transfer Learning model based on VGG19.

*4.1 Custom CNN Model*

**Architecture**

Our custom Convolutional Neural Network (CNN) model is designed to classify handwritten digits from the MNIST dataset. It consists of several layers, each serving a specific purpose:

1. *Input Layer*: This layer accepts grayscale images of size (28, 28, 1).

2. *Convolutional Layers*: We have two convolutional layers with 32 and 64 filters, respectively. Each layer uses a 3x3 kernel and ReLU activation functions to extract features from the input.

3. *Max Pooling Layers*: After each convolutional layer, we apply a max-pooling layer with a 2x2 pool size to reduce the spatial dimensions.

4. *Flatten Layer*: This layer flattens the output from the previous layers into a one-dimensional vector.

5. *Dense Layer*s: We have two fully connected (Dense) layers with 128 neurons each, followed by a dropout layer with a 50% dropout rate to prevent overfitting.

6. Output Layer: The final layer contains 10 neurons, one for each digit (0-9), and uses softmax activation to produce probability scores for each class.

*Compilation*

The model is compiled with the following settings:

- *Loss Function*: Sparse Categorical Cross entropy

- *Optimize*r: Adam

- *Metrics:* Accuracy

*Training*

The training process involves the following key steps:

* - *Dataset Split:* We split the MNIST dataset into training and validation sets (80% training, 20% validation).
* - *Data Preprocessing*: Input pixel values are scaled to the range [0, 1].
* - *Hyper parameters:* We experimented with different hyper parameters, including batch size, learning rate, and number of epochs, to optimize model performance.
* - *Model Evaluation:* We monitored the model's accuracy on the validation set during training to assess its performance.

*4.2 Transfer Learning with VGG19*

*Use of VGG19 for Transfer Learning*

Transfer learning is a technique where we leverage the knowledge gained from a pre-trained model on a different dataset to boost the performance of our task. In this case, we utilized the VGG19 architecture, which was originally trained on the ImageNet dataset, and adapted it for our MNIST digit classification task.

*Custom Layers Added*

To adapt VGG19 for our task, we made the following modifications:

- Removed the original classification layer at the top, as it was designed for ImageNet's 1,000 classes.

- Added a custom Flatten layer to convert the VGG19 output into a one-dimensional vector.

- Added two Dense layers with 128 neurons each, followed by a dropout layer for regularization.

- A final Dense layer with 10 neurons (one for each digit) and softmax activation was added for classification.

*Compilation*

The model was compiled with the same settings as the custom CNN model:

- Loss Function: Sparse Categorical Crossentropy

- Optimizer: Adam

- Metrics: Accuracy

*Training*

The training process for the Transfer Learning model is similar to that of the custom CNN model:

- Dataset Split: We split the MNIST dataset into training and validation sets.

- Data Preprocessing: Input pixel values were scaled to [0, 1].

- Hyper parameters: We fine-tuned hyper parameters such as batch size, learning rate, and number of epochs.

- Model Evaluation: Validation accuracy was monitored to gauge the model's performance during training.

The Transfer Learning model leverages the knowledge encoded in VGG19's convolutional layers, which were trained on diverse image data, to improve its ability to extract meaningful features from the MNIST digit images.

# 5. Model Evaluation

In this section, we will examine how both models, the custom CNN model and the Transfer Learning model based on VGG19, are evaluated on the testing set. We will explore the metrics used, provide classification reports, display confusion matrices, and visualize performance metrics.

*5.1 Evaluation Metrics*

Metrics Used

To assess the models' performance, we employ several key evaluation metrics:

- Accuracy: Accuracy measures the proportion of correctly classified instances out of the total instances in the testing set. It provides an overall view of model performance.

- Precision: Precision quantifies the ratio of true positive predictions to the total positive predictions. It's valuable for scenarios where false positives are costly.

- Recall (Sensitivity): Recall calculates the ratio of true positive predictions to the total actual positive instances. It's essential for scenarios where false negatives are critical.

- F1-Score: The F1-Score is the harmonic mean of precision and recall, offering a balanced assessment of a model's performance.

*5.2 Classification Reports*

Custom CNN Model

We present a classification report for the Custom CNN Model on the testing set, which includes precision, recall, F1-score, and support for each class (digit).

Transfer Learning with VGG19

Similarly, we provide a classification report for the Transfer Learning Model (VGG19) on the testing set, including precision, recall, F1-score, and support for each class.

*5.3 Confusion Matrices*

Custom CNN Model

To visually analyze the model's performance, we present a confusion matrix for the Custom CNN Model. This matrix illustrates the true positive, true negative, false positive, and false negative predictions for each class.

Transfer Learning with VGG19

We also display a confusion matrix for the Transfer Learning Model (VGG19) to visually evaluate its performance on the testing set.

*5.4 Visualization of Performance Metrics*

Custom CNN Model

Performance metrics for the Custom CNN Model are visualized using appropriate plots or tables. This visualization helps in understanding the model's strengths and weaknesses.

Transfer Learning with VGG19

Similarly, we visualize the performance metrics for the Transfer Learning Model (VGG19) to gain insights into its classification capabilities.

# 6. Overfit Analysis

In this section, we will discuss the analysis of overfitting in both the Custom CNN Model and the Transfer Learning Model based on VGG19. We will explain how overfitting is monitored, provide training and validation curves, and describe the steps taken to mitigate overfitting.

*6.1 Monitoring Overfitting*

Custom CNN Model

To monitor overfitting in the Custom CNN Model, we closely examine the training and validation curves. These curves illustrate the model's performance on the training and validation sets during training epochs.

Transfer Learning with VGG19

Similarly, we monitor overfitting in the Transfer Learning Model (VGG19) by analyzing the training and validation curves. These curves provide insights into the model's behavior as it trains.

*6.2 Training and Validation Curves*

Custom CNN Model

We present training and validation curves for the Custom CNN Model. These curves show how various metrics, such as accuracy and loss, evolve over the training epochs. By comparing the curves, we can identify signs of overfitting.

Transfer Learning with VGG19

Likewise, we provide training and validation curves for the Transfer Learning Model (VGG19). These curves offer a visual representation of the model's training progress and help us evaluate potential overfitting.

*6.3 Mitigating Overfitting*

Custom CNN Model

To mitigate overfitting in the Custom CNN Model, we employed the following strategies:

- Dropout: We introduced dropout layers to reduce the risk of overfitting by randomly deactivating some neurons during training.

- Early Stopping: We monitored validation loss and implemented early stopping to halt training when validation loss stopped improving.

Transfer Learning with VGG19

For the Transfer Learning Model (VGG19), we took similar steps to mitigate overfitting:

- Dropout: Dropout layers were included in the custom layers added to the VGG19 architecture.

- Early Stopping: We applied early stopping based on validation loss to prevent overfitting.

*6.4 Results of Overfitting Analysis*

We provide an analysis of the results obtained from monitoring overfitting, including insights gained from the training and validation curves. We also discuss the effectiveness of the mitigation strategies applied and their impact on model performance.

# 7. Conclusion

In conclusion, our digit classification project has yielded insightful results and valuable comparisons between two distinct models: the Custom CNN Model and the Transfer Learning Model based on VGG19. The Custom CNN Model demonstrated commendable accuracy and precision on the testing set, showcasing its ability to classify handwritten digits effectively. However, the Transfer Learning Model, benefiting from the knowledge embedded in VGG19's pre-trained features, exhibited superior accuracy and robust generalization, proving its efficacy in tackling the digit classification task. Throughout our training and evaluation processes, we closely monitored for signs of overfitting and employed techniques such as dropout and early stopping to mitigate this challenge in both models. The transfer learning approach revealed the benefits of leveraging pre-trained convolutional layers, allowing us to achieve remarkable results. As we look to the future, we anticipate further model refinements and explore real-world applications where accurate digit classification holds significant promise. In closing, this project underscores the potential of machine learning models in solving practical image classification tasks and highlights the advantages of transfer learning in enhancing model performance.