CNN Project Report: Handwritten Digit Recognition

Team Member: Alan Xing, Chien-I Chao, Nicholas Rasmussen

Introduction

In this project, we delve into the fascinating world of convolutional neural networks (CNNs) to tackle the task of handwritten digit recognition. Our goal is to build an accurate model that can identify digits (ranging from 0 to 9) from grayscale images. Let's break down the key steps involved in this endeavor:

1. Data Preparation

1.1 Loading the MNIST Dataset

- We start by loading the **MNIST dataset**:
  - The MNIST dataset is a classic benchmark widely used for handwritten digit recognition tasks.
  - It consists of **grayscale images** of handwritten digits (0 to 9).
  - Each image is a **28x28 pixel** representation, making it a total of **784 pixels** per image.

1.2 Preprocessing

- Before feeding the data into our neural network, we perform essential preprocessing steps:
  1. **Normalization**:
     - The pixel values in the original images range from 0 to 255 (8-bit grayscale).
     - To ensure consistent scaling, we normalize the pixel values based on the **maximum pixel value observed in the dataset**.
     - In our case, the maximum pixel value is **16** (unlike the usual 255), which we discovered during exploration.
     - Normalization helps the model converge faster during training.
  2. **Data Splitting**:
     - We divide the dataset into **training** and **testing** sets:
       - **80%** of the data is used for training.
       - **20%** is reserved for testing.
     - This common practice ensures that we evaluate the model's performance on unseen data.

1.3 Data Augmentation for Robust Training

- To enhance the model's robustness, we augment the training data:
  - We introduce **Gaussian noise** to each image:
    - The `add_gaussian_noise` function adds random noise sampled from a Gaussian distribution.
    - This process simulates variations that the model might encounter during real-world scenarios.
    - The noisy images are clipped to ensure pixel values remain within the valid range of [0, 1].
  - The augmented training set now includes both original and noisy images.

1.4 One-Hot Encoding of Labels

- Our digit recognition problem involves **10 classes** (digits from 0 to 9).
- To facilitate model training, we perform **one-hot encoding** on the labels:
  - Each digit label is transformed into a binary vector of length 10.
  - The position corresponding to the true class is set to 1, while others remain 0.
  - For example, if the true label is 5, the one-hot encoded vector is [0, 0, 0, 0, 0, 1, 0, 0, 0, 0].

2. Model Architecture

2.1 Input Layer

- The input layer receives grayscale images of size **8x8x1** (where 1 represents the single channel for grayscale).
- The `Input` layer initializes the network with the specified input shape.

2.2 Feedforward Network

2.2.1 Convolutional Layers

- The Feedforward Network begins with a **Conv2D** layer:
  - It applies 32 filters (also known as kernels) to the input.
  - The filters have a kernel size of **3x3** and use **"same"** padding to maintain spatial dimensions.
  - Batch normalization ensures stable training by normalizing the output.
  - The **ReLU** activation function introduces non-linearity.
  - The output shape is **8x8x32**.
- Next, we apply **max pooling**:
  - A **MaxPooling2D** layer reduces spatial dimensions by a factor of 2.

    - The output becomes **4x4x32**.
- To prevent overfitting, we use **spatial dropout**:
    - **SpatialDropout2D** randomly drops entire feature maps during training.

2.2.2 Additional Convolutional Layers

- We add another Conv2D layer:
    - This time with 64 filters.
    - Batch normalization and ReLU activation are applied.
    - Max pooling reduces the output to **2x2x64**.
- Regular dropout is used:
    - **Dropout** randomly sets a fraction of neurons to zero during training.
- Finally, we add a third Conv2D layer:
    - 128 filters are applied.
    - Batch normalization and ReLU activation are again used.

3. Max Pooling Layers in Convolutional Neural Networks

3.1 Purpose of Max Pooling

- **Max pooling** is a downsampling operation commonly used in convolutional neural networks (CNNs).
- Its primary purpose is to reduce the spatial dimensions of feature maps while retaining essential information.
- By doing so, max pooling helps control the model's complexity, reduces computation, and introduces translation invariance therefore contributing to the hierarchical representation learning in CNNs.

3.2 Consistent MaxPooling2D Layers

- Throughout your feedforward network, you've consistently applied **MaxPooling2D** layers.
- These layers operate on 2D feature maps (usually after convolutional layers) and perform downsampling.

3.3 Key Parameters

3.3.1 Pooling Kernel (Filter) Size

- The **pooling kernel** (also known as the filter) determines the size of the local region over which max pooling is performed.
- In our case, we used a **(2x2)** pooling kernel.
- This means that for each 2x2 region in the feature map, the maximum value is selected.

3.3.2 Stride

- The **stride** specifies the step size when sliding the pooling kernel across the feature map.
- In your case, the stride is also **(2x2)**.
- This means that the pooling kernel moves by 2 units (both horizontally and vertically) at each step.

3.3.3 Effect on Dimensions

- The stride of **(2x2)** reduces both the **x** and **y** dimensions of the feature map by half.
- For example:
    - If the original feature map size was **(4x4)**, max pooling with a **(2x2)** kernel and stride would result in a new feature map of size **(2x2)**.
    - The final output is **1/4th** the size of the original feature map.

3.4 Visual Representation

- Imagine a feature map with values:
- `1  2  3  4`
- `5  6  7  8`
- `9 10 11 12`
- `13 14 15 16`
- Applying max pooling with a **(2x2)** kernel and stride:
    - The first pooling region is **[1, 2, 5, 6]**, and the maximum value is **6**.
    - The second pooling region is **[3, 4, 7, 8]**, and the maximum value is **8**.
    - The third pooling region is **[9, 10, 13, 14]**, and the maximum value is **14**.
    - The fourth pooling region is **[11, 12, 15, 16]**, and the maximum value is **16**.
- The resulting feature map becomes:
- `6  8`
- `14 16`

4. Fully Connected Layer (Dense Layer)

4.1 Flattening the Features

- After the convolutional layers extract relevant features from the input images, we arrive at a 3D matrix.
- Specifically, the output shape from the last convolutional layer is **(2x2x128)**.
- To prepare these features for further processing, we **flatten** them into a 1D vector.
- The flattened vector has a size of **512** (since 2x2x128 = 512).

4.2 Dense Layer with 64 Neurons

- The flattened features are then passed to a **Dense** layer (also known as a fully connected layer).
- This layer consists of **64 neurons**:
    - Each neuron is connected to every element in the flattened feature vector.
    - The connections are weighted, and the weights are learned during training.
    - The purpose of this layer is to capture complex patterns and relationships among the features.
- **ReLU Activation**:
    - The output of each neuron is computed as the weighted sum of inputs.
    - The ReLU (Rectified Linear Unit) activation function introduces non-linearity.
    - ReLU(x) = max(0, x)
    - It ensures that negative values are set to zero, allowing the network to learn complex representations.
- **Regular Dropout**:
    - Dropout is applied to regularize the model and prevent overfitting.
    - During training, a fraction of neurons (specified by the dropout rate) is randomly set to zero.
    - This encourages the network to rely on different subsets of neurons during each forward pass.

4.3 Output Layer

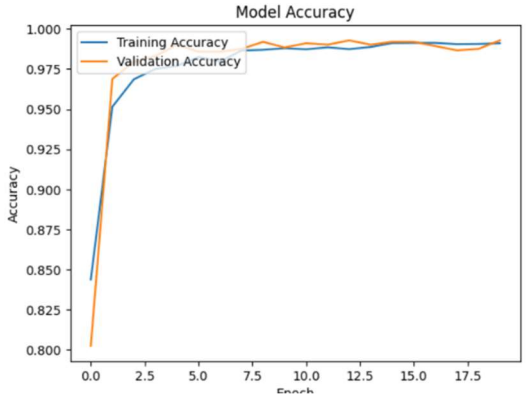
- The final layer in the neural network architecture is the **output layer**.
- It consists of **num_classes** neurons (in this case, 10 for digit classification).
- The **softmax** activation function is applied to the raw scores produced by these neurons:
    - Softmax converts the raw scores into a probability distribution across the classes.
    - The output represents the likelihood of each class.
    - The class with the highest probability is the predicted class.

5. Model Performance

5.1 Plots and Confusion Matrix

5.1.1 Training and Validation Accuracy

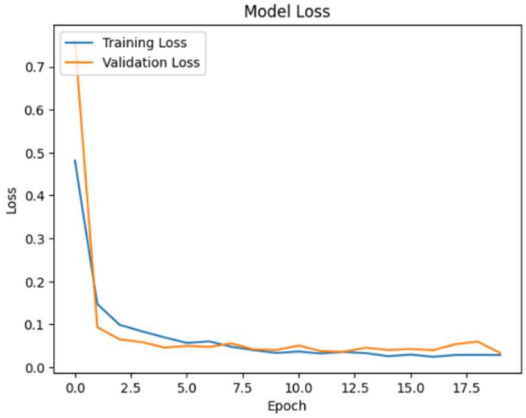
The plot below shows the training accuracy and validation accuracy over different epochs. It's essential to monitor both curves to assess how well the model is learning from the data.



- The **Training Accuracy** curve represents how well the model performs on the training data during each epoch.
- The **Validation Accuracy** curve shows the model's performance on a separate validation dataset (not used during training). It helps us understand if the model is overfitting or generalizing well.

5.1.2 Training and Validation Loss

The next plot illustrates the training loss and validation loss:

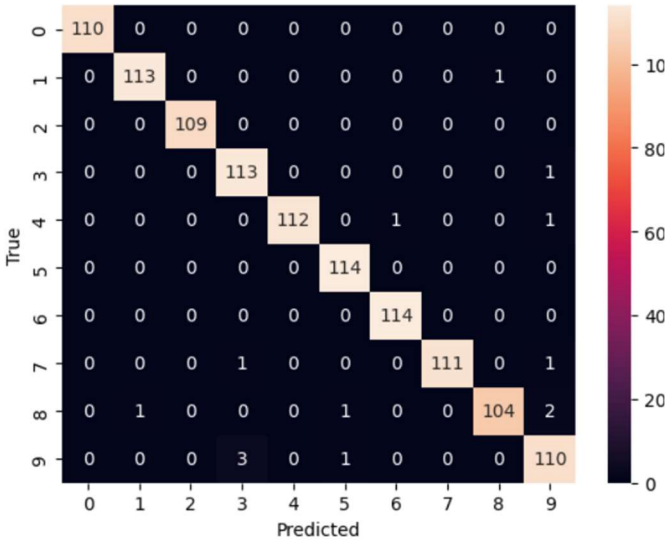


- The **Training Loss** curve represents the error (loss) on the training data during each epoch. Lower values indicate better performance.
- The **Validation Loss** curve shows the error on the validation dataset. We want this to be as low as possible without overfitting.

5.1.3 Confusion Matrix

A confusion matrix provides insights into the model's performance across different classes. Each row represents the true class, while each column represents the predicted class. The diagonal elements represent correct predictions, and off-diagonal elements represent misclassifications.

Here's an example of a confusion matrix:



- The numbers in each cell indicate the count of samples.
- The diagonal elements (top-left to bottom-right) represent correct predictions.
- Off-diagonal elements represent misclassifications.

5.2 K-fold Cross-Validation

- We start by initializing lists to store metrics across different folds:
- Next, we set up **k-fold cross-validation** using `KFold` with 5 splits (folds). This technique divides the dataset into k subsets (folds) and iteratively trains and evaluates the model on different subsets.
- For each fold:
    - We reset the model weights to ensure consistent initialization.
    - Split the data into training and test sets using the indices provided by `KFold`.
    - Create an augmented training set by adding noise to each image.
    - Convert the lists of augmented data to numpy arrays.
    - Train the model on the augmented training data for 20 epochs, using the validation data for evaluation.

5.2.1 Model Evaluation and Metrics

- After training, we make predictions on the test set (`X_test`).
- The predicted classes are obtained by selecting the class with the highest probability from the model's output.

- We compute a **classification report**:
    - This report includes precision, recall (sensitivity), F1-score, and support for each class.
    - The target names are provided as a list of class names (e.g., "Class0," "Class1," etc.).

5.2.2 Parsing the Classification Report

- We parse the classification report to extract relevant metrics for each class:
    - For each line (excluding header and footer lines), we split the line to extract class name, precision, recall, and F1-score.
    - These metrics are stored in the `metrics` dictionary for further analysis.

5.3 Metrics Analysis Across Folds

5.3.1. Initialization and Data Preparation

- We start by extracting the precision, recall (sensitivity), and F1-score values from the `metrics` dictionary. These values were computed for each class during k-fold cross-validation.

5.3.2 Calculating Metrics Statistics

5.3.2.1 Average Metrics

- **Average Precision**:
    - We compute the mean precision across all classes.
    - Precision measures the proportion of true positive predictions among all positive predictions.
- **Average Recall (Sensitivity)**:
    - We calculate the mean recall across all classes.
    - Recall quantifies the proportion of true positive predictions among all actual positive instances.
- **Average F1-score**:
    - We compute the mean F1-score across all classes.
    - The F1-score is the harmonic mean of precision and recall.

5.3.2.2 Range of Metrics

- **Range of Precision**, **Range of Recall**, and **Range of F1-score**:
    - These values represent the difference between the maximum and minimum metric values across different folds.
    - A larger range indicates greater variability in performance.

5.3.2.3 Standard Deviation of Metrics

- **Standard Deviation of Precision**, **Standard Deviation of Recall**, and **Standard Deviation of F1-score**:
    - These values quantify the spread or dispersion of metric values around the mean.
    - A higher standard deviation suggests more variability in performance.

5.3.3 Summary

- The calculated metrics provide insights into the model's consistency and performance across different folds.
- We ese these statistics to assess the stability and reliability of your model.

6. Documentation and Analysis

6.1 Component Explanation

- The above document provides detailed explanations of our CNN implementation.

6.2 Code Readability

- We've included comments in the code to enhance readability.
- Clear and well-documented code ensures that others can understand and reproduce our work.

7. GitHub

- We will follow the 2AI GitHub page.
- Each separate member will upload the code to their repository.

Conclusion

This project allowed us to learn and apply CNNs to handwritten digit recognition. While our group assignment may be simpler than industry applications, the experience gained is valuable. CNNs and computer vision continue to play a crucial role in various sectors, and this project contributes to our understanding of these technologies.