Building a Convolutional Neural Network from Scratch

Ta Quang Minh

June 12, 2024

Abstract

This report presents the development of a Convolutional Neural Network (CNN) built from scratch. The project involves designing, implementing, and evaluating a CNN model for image classification tasks. The report details the theoretical background, implementation steps, experimental setup, and results.

Contents

Chapter 1

Introduction

1.1 Background

Convolutional Neural Networks (CNNs) have revolutionized the field of computer vision, enabling significant advances in image classification, object detection, and image generation. This project aims to explore the fundamental concepts behind CNNs by building one from the ground up.

1.2 Objectives

The primary objectives of this project are:

- To understand the architecture and working principles of CNNs.
- To implement a CNN from scratch with only standard functions without relying on deep learning or linear algebra libraries.
- To evaluate the performance of the implemented CNN on a benchmark dataset.

Chapter 2

Theoretical Background

2.1 Convolutional Layer

A convolutional layer is one of the primary building blocks of a Convolutional Neural Network (CNN). It primarily functions to extract various features from the input data through the application of filters (or kernels), as well as to reduce the size of the inputs.

There are some fundamental parameters that define the operation of a convolutional layer, which include:

- **Kernel Size:** A filter is a small matrix used to apply effects such as edge detection...

  Each filter is characterized by its depth, height, and width, which should match the depth of the input data for the first layer or the depth of the previous layer's output.

  The size of the filter affects the amount of information each filter can capture. A larger kernel will capture more information in one receptive field but may miss finer details, whereas a smaller kernel captures less information but is better for detecting finer details. Common sizes include 3×3, 5×5, and 7×7.

- **Number of Kernels:** The number of filters in a convolutional layer determines the number of unique features that can be extracted.

  These filters are initialized randomly when the convolutional layer is created, and they are updated gradually to fit the data as the model trains.

- **Stride:** Stride dictates the movement of the filter across the input.

  For example, a stride of 1 moves the filter one pixel at a time, while a stride of 2 moves it two pixels. A larger stride results in a smaller output dimension.

- **Padding:** To handle the reduction in dimensionality and preserve the original size of the input, padding can be applied around the edges of the input.

  There are several options for padding: padding with a fix number like 0, reflective or replicate padding. The number of rows/columns padded to the border can also be a parameter to specify.

How it works

The convolutional operation involves sliding each filter across the entire input volume, computing the dot product between the entries of the filter and the input at any position.

For multichannel input, for each filter, this convolution operation is applied for each corresponding filter slice-input slice pair and then sum up the resulting matrices to produce an output.

In a more purely mathematical definition, this operation is rather called "cross-correlation" rather than convolution. In a true convolution, the filter (or kernel) is first flipped both horizontally and vertically before being slid over the input feature map. This flipping process is a defining characteristic of convolution operations as used in fields like signal processing. However, in CNNs, the filters are applied directly to the input without this flipping step. Despite this difference, the term "convolution" is conventionally used due to historical reasons and the functional similarity in terms of capturing spatial hierarchies and patterns in the input data.

The collective outputs from all filters stacking together then form the output volume. The depth of the output volume is equal to the number of filters, while the spatial dimensions of the output volume are determined by the kernel size, stride, and padding used.

2.2 Pooling Layer

The pooling layer is used primarily to reduce the spatial dimensions (width and height) of the input volume for the subsequent layers. By reducing the number of parameters and computations in the network, it helps to control overfitting.

It operates independently on every depth slice of the input and resizes it spatially, commonly using the MAX or AVERAGE operation.

Max pooling involves selecting the maximum value from a group of values in a filter-sized patch of the input data. For instance, with a 2×2 filter and a stride of 2, max pooling will consider each 2×2 square block of the input data, output the maximum value, and discard all other values. The key parameters include:

- **Filter Size:** Common filter sizes are 2×2 or 3×3. Larger filters increase the downsampling rate.
- **Stride:** Typically set equal to the filter size to prevent overlap; for a 2×2 filter, a stride of 2 is common.
- **Padding:** Usually, padding is not used with max pooling.

Difference between Max Pooling and Average Pooling:

While max pooling selects the maximum value from the filter region, average pooling calculates the average of the values within the filter region.

- **Feature Sensitivity:** Max pooling is generally more sensitive to the most prominent features, making it more effective in feature detection. Average pooling, however, can smooth out features since it takes into account all values equally, which might be beneficial in reducing noise but at the cost of losing important feature sensitivity.
- **Impact on Model Performance:** Max pooling tends to perform better in tasks where the detection of strong, distinctive features is crucial, such as in image recognition. Average pooling can be more suitable for tasks where preserving the background information is more important.

2.3 Fully Connected Layer

Fully connected (FC) layers in a CNN have identical structure to layers in a feed forward neural network. It is often employed at the tail end of the network, after convolutional and pooling layers have been applied. This is to reduce the number of neurons have to use in these particular layers, which are the most computational costly.

After feature extraction through successive convolutional and pooling layers, the role of the fully connected layers is to interpret these features and map them to the final output such as class scores in classification tasks.

In a fully connected layer, neurons have connections to all activations in the previous layer, as opposed to convolutional layers where neurons are only connected to local regions. By flattening the output of the previous layers (convolutional or pooling layers), each neuron in a fully connected layer can see all the data that represents high-level features extracted by the convolutional and pooling layers.

Each neuron in a fully connected layer has a weight associated with every input value and a bias. The outputs of the fully connected layer are computed by a matrix multiplication followed by a bias offset and typically followed by a non-linear activation function.

2.4 Activation Functions

Activation functions introduce non-linearity into the network, enabling it to learn and model more complex patterns in the data. Activation functions are applied to the output of the convolution layer and the fully connected layer (often not applied to the output of pooling layer).

Some common activation function may include:

- **Sigmoid:** The sigmoid function is given by $f(x) = \frac{1}{1+e^{-x}}$. It squashes its input to a range between 0 and 1, which is interpretable as a probability. It is one of the first activation functions used in neural network and is still often used in the output layer of binary classification problems. However, its usage in hidden layers has declined because it can cause gradients to vanish during back-propagation, as gradients are very small away from the center part of the function.

- **ReLU (Rectified Linear Unit):** The ReLU function is defined as $f(x) = \max(0, x)$. It outputs the input directly if it is positive; otherwise, it outputs zero. ReLU is particularly popular in CNNs due to its computational efficiency and its ability to reduce the likelihood of the vanishing gradient problem. By allowing the back-propagation of gradients only through positive elements, it maintains the activation of the neuron only when it is beneficial for learning.

- **Tanh (Hyperbolic Tangent):** The tanh function is represented as $f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$. Similar to the sigmoid, it squashes its inputs to a range between -1 and 1. This zero-centered property can make the optimization easier in some cases, but like sigmoid, it also suffers from vanishing gradients.

- **Leaky ReLU:** To address some issues of the ReLU function, such as neurons dying (i.e., not activating across any data point in the dataset), the Leaky ReLU function is used. It is defined as $f(x) = \max(\alpha x, x)$ where α is a small constant (e.g., 0.01). This small slope ensures that even when the input is less than zero, the neuron will still have some gradient through which learning can continue.

- **Softmax:** Used primarily in the output layer of a multinomial classification CNN, the softmax function converts logits, the raw output scores in neural networks, into probabilities by taking the exponential of each output and then normalizing these values by dividing by the sum of all the exponentials. This output can then be directly interpreted as a probability distribution over the classes. It is mathematically represented as $f(x_i) = \frac{e^{x_i}}{\sum_k e^{x_k}}$ where x is the input vector, and i and k index its elements.

2.5 Loss Functions

Loss functions quantify the difference between the predicted outputs and the actual target values. The goal of training is to minimize this loss, which ideally represents the error in predictions made by the network.

Types of Loss Functions:
For classification problem, the common loss are:

- **Cross-Entropy Loss:** This is the most widely used loss function for classification problems, where the objective is to predict the correct class out of multiple classes. Mathematically, it is defined for a single sample as $L = -\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$, where M is the number of classes, y is a binary indicator (0 or 1) if class label c is the correct classification for observation o, and p is the predicted probability of observation o being of class c. Cross-entropy loss measures the performance of a classification model whose output is a probability value between 0 and 1. For binary classification, it simplifies to $L = -y\log(p) - (1 - y)\log(1-p)$.

- **Hinge Loss:** Commonly used for "maximum-margin" classification, most notably for support vector machines. For CNNs, it's particularly useful in binary classification tasks. Hinge loss is defined as $L = \max(0, 1 - t_o)$, where $t_o = y \cdot f(x)$, $y \in \{-1, 1\}$ is the label, and $f(x)$ is the predicted score. Hinge loss aims to ensure that the correct class score is not only higher than the incorrect class scores but exceeds them by a certain margin.

- **Categorical Cross-Entropy:** It is a variant of Cross-Entropy that is used when the target labels are one-hot encoded, meaning each label is a vector comprising 0s and a single 1 indicating the correct class. It computes the loss by taking the cross-entropy between the predicted class probabilities and the one-hot encoded label.

For regression problem, the common loss are:

- **Mean Squared Error (MSE) Loss:** Although more common in regression tasks, MSE can be used in CNNs for predicting continuous output values. It is calculated as the average of the squares of the differences between the predicted and actual values, defined by $L = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$, where y_i is the actual value and $\hat{y}_i$ is the predicted value, and N is the number of samples. MSE loss penalizes larger errors more than smaller ones due to squaring the error term.

- **Huber Loss:** Huber loss combines the best properties of MSE and MAE. It is quadratic for small errors and linear for large errors, making it less sensitive to outliers. It is defined as:

$$L_{\text{Huber}} = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq \delta \\ \delta(|y - \hat{y}| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

  where δ is a threshold parameter.

Role of Loss Functions in Training CNNs:

Loss functions guide the optimization algorithms on how to adjust the weights of the network to minimize the prediction error. The choice of loss function can significantly affect the convergence speed and the quality of the final model. During backpropagation, the gradient of the loss function is computed to update the weights, which incrementally decreases the loss, leading the model to improve its accuracy over iterations.

2.6 Backpropagation in Convolutional Neural Networks

Backpropagation is a fundamental mechanism used for training Convolutional Neural Networks (CNNs) and other types of neural networks. It involves the iterative process of adjusting the weights of the network to minimize the loss function.

Overview of Backpropagation:

Backpropagation in CNNs operates through the following key steps:

1. **Forward Pass:** The network processes the input data layer by layer from the input layer to the output layer. Each layer applies its weights and biases to the input data, passes through an activation function, and produces an output that serves as the input for the next layer.

2. **Loss Calculation:** Once the forward pass reaches the output layer, the network computes the output error using a loss function. This function quantifies the difference between the predicted output and the actual target values.

3. **Backward Pass:** The core of backpropagation begins here. The error calculated at the output is propagated back through the network, layer by layer. Each layer calculates the gradient of the loss function with respect to its weights and biases. This gradient tells us how to change the weights to decrease the loss.

Mathematical Mechanism:

The mathematical principle behind backpropagation is the chain rule from calculus. For a given weight w in the network, the change needed to reduce the error is proportional to the partial derivative of the loss function L with respect to w, denoted as $\frac{\partial L}{\partial w}$.

- **Gradient Calculation:** For each neuron, compute the gradient of the loss with respect to the input. This value is stored to use in the future because it correspond with the gradient of the loss with respect to the output of the previous layer (which we will need to use in the backward pass)

  Then, the gradient of the loss with respect to each weight is calculated as the product of the gradient of the loss with respect to the output of the neuron and the derivative of the neuron output with respect to the weight.

- **Weight Update:** Once the gradients are computed, the weights are updated using an optimization algorithm such as Stochastic Gradient Descent (SGD) or Adam.

  The update rule typically looks like $w = w - \eta\frac{\partial L}{\partial w}$, where η is the learning rate, a small positive number that determines the size of the step to take in the direction that minimally reduces the loss.

Considerations in CNN Backpropagation:
In CNNs, the backpropagation algorithm must account for the specific structure of convolutional and pooling layers:

- **Convolutional Layers:** The gradients of the loss function are propagated back through the convolution operations.

  The weight gradients for the convolution filters are computed by applying the gradient of the loss with respect to the output feature map to the input feature map regions corresponding to each filter application.

- **Pooling Layers:** For max pooling layers, the gradient is only propagated back to the neurons that produced the maximum value during the forward pass. For average pooling, the gradient is distributed equally to all neurons in the input region.

  Since pooling layers have no learnable parameter, there will not be any weights being updated.

Chapter 3

Implementation

3.1 Building the CNN

In this project, only standard functions and methods are employed. I did not utilize deep learning frameworks like TensorFlow or linear algebra libraries like NumPy. The goal was to use the minimum number of external libraries to complete the tasks.

The source code for the project is submitted alongside this report in a separate file.

Below, I provide a high-level description of how I implemented the model.

3.1.1 Utility Functions

Several utility functions were defined for the network, including:

- **Activation Functions**: ReLU, softmax, and their derivatives.
- **Loss Function**: Categorical cross-entropy loss and its derivative when used with softmax activation.
- **Other Functions**: Functions to reshape matrices, perform element-wise multiplication, etc.

3.1.2 Convolution Layer

This class is initialized with five arguments: filter size, number of filters, number of channels (for the input), stride, and padding size.

An in-class padding method is implemented to apply zero-padding according to the specified padding size.

The input to this layer is expected to have the shape (batch size, number of channels, height, width).

Initialization

The initial weights of each filter is assigned via Xavier initialization with the standard normal distribution. The motivation for this is so that the weights balances the forward and backward propagations of signals, thereby mitigating the issues associated with vanishing or exploding gradients and promoting more stable training dynamics. In mathematical terms, the Xavier initialization formula for a given weight matrix W is:

$$W \sim \mathcal{N}\left(0, \frac{1}{n_{\text{in}}}\right)$$

where $\mathcal{N}$ denotes a Gaussian (normal) distribution and with n_{in} denote the input dimension.

Forward method

In the forward method, the filters will be slide and convolution calculated to return a feature map to be feed to the next layer.

Given the input tensor $\mathbf{X}$ with dimensions (N, C, H, W) where N is the batch size, C is the number of input channels, H is the height, and W is the width. Let $\mathbf{W}_f$ be the filter weights with dimensions (F, C, f_h, f_w) where F is the number of filters, and f_h and f_w are the filter height and width respectively. The stride and padding are denoted as s and p, respectively.

The output dimensions H_{out} and W_{out} are computed as:

$$H_{out} = \left\lfloor \frac{H + 2p - f_h}{s} \right\rfloor + 1 \tag{3.1}$$

$$W_{out} = \left\lfloor \frac{W + 2p - f_w}{s} \right\rfloor + 1 \tag{3.2}$$

The output tensor $\mathbf{Y}$ with dimensions (N, F, H_{out}, W_{out}) is calculated as:

$$Y_{n,f,i,j} = \sum_{c=0}^{C-1} \sum_{h=0}^{f_h-1} \sum_{w=0}^{f_w-1} X_{n,c,i\cdot s+h,j\cdot s+w} \cdot W_{f,c,h,w} \tag{3.3}$$

where:

$$
\begin{aligned}
n &: \text{batch index},\\
f &: \text{filter index},\\
i &: \text{output height index},\\
j &: \text{output width index},\\
c &: \text{input channel index},\\
h &: \text{filter height index},\\
w &: \text{filter width index}.
\end{aligned}
$$

Backward method

In the backward method, the gradient of loss with respect to the input, and the gradient of loss with the weights are being calculated. The former is returned for being back-propagated to the next layer, and the later being used to update the weight via gradient descent.

Given the gradient of the loss with respect to the output of the convolutional layer $\frac{\partial L}{\partial \mathbf{Y}}$ with dimensions (N, F, H_{out}, W_{out}), where N is the batch size, F is the number of filters, H_{out} is the output height, and W_{out} is the output width. Let $\mathbf{X}_{\text{pad}}$ be the padded input tensor and $\mathbf{W}_f$ be the filter weights. The gradient with respect to the filter weights $\frac{\partial L}{\partial \mathbf{W}_f}$ and the gradient with respect to the input $\frac{\partial L}{\partial \mathbf{X}}$ are computed as follows:

Gradient with Respect to Filter Weights

$$\frac{\partial L}{\partial W_{f,c,h,w}} = \sum_{n=0}^{N-1} \sum_{i=0}^{H_{out}-1} \sum_{j=0}^{W_{out}-1} \frac{\partial L}{\partial Y_{n,f,i,j}} \cdot X_{\text{pad},n,c,i\cdot s+h,j\cdot s+w} \tag{3.4}$$

Gradient with Respect to Input

$$\frac{\partial L}{\partial X_{n,c,x,y}} = \sum_{f=0}^{F-1} \sum_{i=0}^{H_{out}-1} \sum_{j=0}^{W_{out}-1} \frac{\partial L}{\partial Y_{n,f,i,j}} \cdot W_{f,c,x-i\cdot s,y-j\cdot s} \tag{3.5}$$

where:

$$\begin{aligned}
n &: \text{batch index,}\\
c &: \text{input channel index,}\\
x &: \text{input height index,}\\
y &: \text{input width index,}\\
f &: \text{filter index,}\\
i &: \text{output height index,}\\
j &: \text{output width index.}
\end{aligned}$$

The filter weights are updated using the gradients and a learning rate α:

$$W_{f,c,h,w} \leftarrow W_{f,c,h,w} - \alpha \cdot \frac{\partial L}{\partial W_{f,c,h,w}} \tag{3.6}$$

If padding $p > 0$, the gradient with respect to the input $\frac{\partial L}{\partial \mathbf{X}}$ needs to be trimmed to remove the padding.

3.1.3 Pooling layer

In the project, I only implement max-pooling. The input to this layer is expected to have the shape (batch size, nummber of channel, height, width)

Forward method

The forward phase just consist of sliding the filter to capture the max value to reduce the size of the input.

Given the input tensor $\mathbf{X}$ with dimensions (N, C, H, W) where N is the batch size, C is the number of channels, H is the height, and W is the width. Let k be the size of the pooling filter. The stride is equal to the size of the pooling filter. The output dimensions H_{out} and W_{out} are computed as:

$$H_{out} = \left\lfloor \frac{H}{k} \right\rfloor \tag{3.7}$$

$$W_{out} = \left\lfloor \frac{W}{k} \right\rfloor \tag{3.8}$$

The output tensor $\mathbf{Y}$ with dimensions (N, C, H_{out}, W_{out}) is calculated as:

$$Y_{n,c,i,j} = \max_{0 \le m < k} \max_{0 \le n < k} \left(X_{n,c,i \cdot k+m, j \cdot k+n} \right) \tag{3.9}$$

where:

$$\begin{aligned} n &: \text{batch index,} \\ c &: \text{channel index,} \\ i &: \text{output height index,} \\ j &: \text{output width index,} \\ m &: \text{pooling filter height index,} \\ n &: \text{pooling filter width index.} \end{aligned}$$

Backward method

In the backward phase, no parameter need to be update, but the gradient of loss with respect to the input was calculate to be back-propagated to the next layer.

Given the gradient of the loss with respect to the output of the max pooling layer $\frac{\partial L}{\partial \mathbf{Y}}$ with dimensions (N, C, H_{out}, W_{out}), where N is the batch size, C is the number of channels, H_{out} is the output height, and W_{out} is the output width. Let k be the size of the pooling filter. The gradient with respect to the input tensor $\frac{\partial L}{\partial \mathbf{X}}$ is computed as follows:

$$\frac{\partial L}{\partial X_{n,c,i\cdot k+m,j\cdot k+n}} = \begin{cases} \frac{\partial L}{\partial Y_{n,c,i,j}} & \text{if } (m,n) = \text{argmax}_{0\leq m'<k, 0\leq n'<k}\left(X_{n,c,i\cdot k+m',j\cdot k+n'}\right) \\ 0 & \text{otherwise} \end{cases} \tag{3.10}$$

where:

$$\begin{aligned} n &: \text{batch index,} \\ c &: \text{channel index,} \\ i &: \text{output height index,} \\ j &: \text{output width index,} \\ m &: \text{pooling filter height index,} \\ n &: \text{pooling filter width index.} \end{aligned}$$

3.1.4 Fully connected layer

This layer architecture is similar to that of feed forward neural network. The input to this layer is expected to have the shape (batch size, number of input neurons), therefore before inputs are being pushed to the first fully connected layer, it need to be flatten first.

Initialization

I initialize the weights also with Xavier method on standard Gaussian distribution.

Forward method

Given the input vector $\mathbf{x}$ with dimensions (N, D_{in}), where N is the batch size and D_{in} is the input dimension. Let $\mathbf{W}$ be the weight matrix with dimensions (D_{in}, D_{out}), where D_{out} is the output dimension, and $\mathbf{b}$ be the bias vector with dimensions (D_{out}).

The output vector $\mathbf{y}$ with dimensions (N, D_{out}) is calculated as:

$$\mathbf{y}_n = \mathbf{x}_n \mathbf{W} + \mathbf{b} \tag{3.11}$$

Backward method

Given the gradient of the loss with respect to the output of the fully connected layer $\frac{\partial L}{\partial \mathbf{y}}$ with dimensions (N, D_{out}), where N is the batch size and D_{out} is the output dimension.

The gradient with respect to the input $\frac{\partial L}{\partial \mathbf{x}}$ is computed as:

$$\frac{\partial L}{\partial \mathbf{x}_n} = \frac{\partial L}{\partial \mathbf{y}_n} \mathbf{W}^T \tag{3.12}$$

Expanding the equation for each element $\frac{\partial L}{\partial x_{n,i}}$:

$$\frac{\partial L}{\partial x_{n,i}} = \sum_{j=1}^{D_{out}} \frac{\partial L}{\partial y_{n,j}} W_{i,j} \tag{3.13}$$

where:

$$\begin{aligned} n &: \text{batch index,} \\ i &: \text{input dimension index,} \\ j &: \text{output dimension index.} \end{aligned}$$

The gradient with respect to the weights $\frac{\partial L}{\partial \mathbf{W}}$ is computed as:

$$\frac{\partial L}{\partial W_{i,j}} = \sum_{n=1}^{N} x_{n,i} \frac{\partial L}{\partial y_{n,j}} \tag{3.14}$$

The gradient with respect to the biases $\frac{\partial L}{\partial \mathbf{b}}$ is computed as:

$$\frac{\partial L}{\partial b_j} = \sum_{n=1}^{N} \frac{\partial L}{\partial y_{n,j}} \tag{3.15}$$

The weights and biases are updated using the gradients and a learning rate α:

$$W_{i,j} \leftarrow W_{i,j} - \alpha \frac{\partial L}{\partial W_{i,j}} \tag{3.16}$$

$$b_j \leftarrow b_j - \alpha \frac{\partial L}{\partial b_j} \tag{3.17}$$

3.2 Dataset Preparation

To verify the accuracy of my code, I utilized the MNIST dataset to classify images of handwritten digits ranging from 0 to 9.

Given that my model is implemented using Python lists without optimized data structures and algorithms, its performance is relatively slow. Therefore, I selected this dataset due to its small image size.

For demonstration purposes, I used only 350 out of the 60,000 available training images for training (300 for actual training, 50 for validation) and 100 out of the 10,000 available test images for testing.

3.3 Model Architecture

Due to the slow performance of my code and limited resources on Colab, I built a small model with three layers.

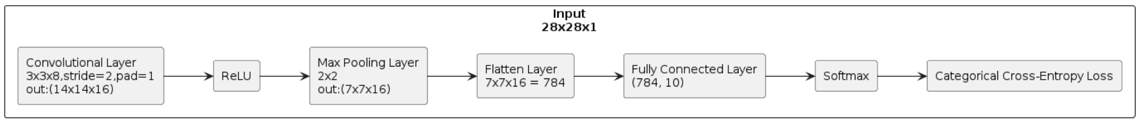

Figure 3.1: SimpleCNN Architecture

The input image dimensions are 28x28x1.

The first layer is a convolutional layer with 16 kernels of size 3x3, a stride of 2, and padding of 1. The output of this layer has dimensions 14x14x16.

This is followed by a ReLU activation function.

The next layer is a Max Pooling layer with a size of 2 and a stride of 2, effectively reducing the input size by half and producing an output of size 7x7x16.

This feature map is then flattened into a 1D vector and passed to the fully connected layer. This layer has 784 input neurons and 10 output neurons, representing the 10 classes.

The output of the fully connected layer is finally processed by a softmax activation function to provide the probabilities for each class.

3.4 Training and Testing

As previously mentioned, I trained the model on 300 samples from the MNIST dataset, used another 50 samples for validation, and 100 samples for testing.

Due to the small size of the training set, I did not divide it into mini-batches. Instead, I used the entire training set for each batch. The chosen learning rate was 0.07, and the number of training epochs was 40. The training phase took approximately 15 minutes on a Colab CPU.

3.5 Results and Discussion

3.5.1 Training Results

Using the previously defined training setup, the following results were obtained after the training phase:

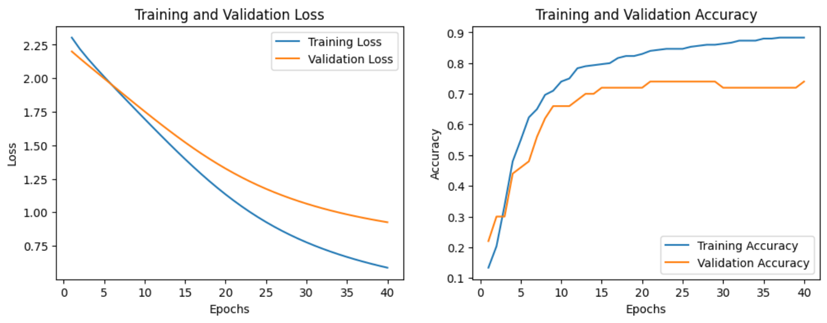

Figure 3.2: Training and Validation Loss and Accuracy Curves

In the early epochs, both training and validation losses decrease steadily, indicating that the model is learning as expected.

Around epoch 15, the validation accuracy begins to plateau even though the losses for both training and validation continue to decrease, albeit at a slower rate. This suggests the onset of overfitting, which is understandable given the small size of the training set.

The loss curves for both training and validation are smooth, without significant fluctuations, indicating that the chosen learning rate was appropriate.

3.5.2 Testing Results

The precision, recall, and accuracy of the model in predicting labels for the test set after training are as follows:

Class	Precision	Recall	F1-Score	Support
0	1.00	0.80	0.89	10
1	0.83	1.00	0.91	15
2	1.00	0.70	0.82	10
3	0.80	1.00	0.89	8
4	0.92	0.61	0.73	18
5	0.83	0.45	0.59	11
6	0.70	1.00	0.82	7
7	1.00	0.88	0.93	8
8	0.67	0.57	0.62	7
9	0.38	1.00	0.55	6
Accuracy			0.78	100
Macro avg	0.81	0.80	0.77	100
Weighted avg	0.84	0.78	0.78	100

The accuracy of the model on the test set is approximately 78%, which is reasonable considering the limited and not carefully prepared training set, as well as the simplicity of the model.

Certain classes, specifically the digits "5", "8", and "9", have particularly low F1 scores.

Finally, the model was tested on some specific images outside the training set. The results are shown below.

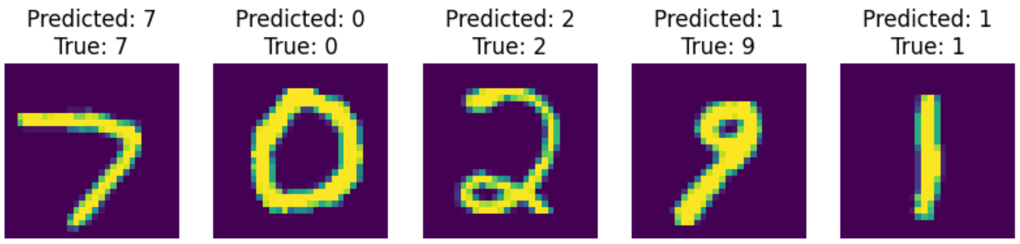

Figure 3.3: Class prediction for some images

3.5.3 Discussion

The objective of this project was to build a simple convolutional neural network for educational purposes and to understand the underlying algorithms. Consequently, state-of-the-art performance was not anticipated. In terms of correctly training on the data and making predictions, albeit with some errors, the model has achieved its intended goal.

For future improvements, the model could benefit from better-optimized code and parallel computing. This would allow for the addition of more layers to the network and enable training with larger datasets within a reasonable time frame. Implementing regularization techniques and dropout could also help reduce overfitting.

Chapter 4

Conclusion

In this project, I successfully implemented a Convolutional Neural Network (CNN) from scratch without the aid of any external libraries such as NumPy. This endeavor involved creating the fundamental components of a CNN, including convolutional layers, activation functions, pooling layers, and fully connected layers, all implemented using basic Python constructs.

The CNN was trained on a small dataset, and although the training process was significantly slower compared to implementations utilizing optimized libraries, the model was able to learn and make predictions. The final accuracy, while not as high as that achieved by models leveraging sophisticated frameworks and hardware acceleration, demonstrates the functionality and correctness of the implemented CNN.

Overall, this project has been a valuable educational exercise in understanding and applying the core principles of Convolutional Neural Networks from a foundational perspective, reinforcing the theoretical knowledge through practical application.