Project Report: Neural Network Implementation in C++

Esen Dashnyam

December 20, 2024

I. Introduction

Background and Motivation

The impetus for this project arises from the growing need for customized neural network implementations tailored to specific applications. While established frameworks like TensorFlow and PyTorch are widely used, building neural networks from scratch offers profound insights into the foundational mathematics and algorithms of deep learning. This project not only reinforces object-oriented programming (OOP) principles in C++ but also deepens the understanding of neural network mechanics by constructing them manually.

Problem Statement

The primary objective of this project is to develop a Multi-Layer Perceptron (MLP) in C++ utilizing OOP methodologies. The MLP is designed to classify handwritten digits from the MNIST dataset. Key challenges addressed include the dynamic construction of network architectures and the implementation of robust training and evaluation mechanisms.

II. Implementation

Foundational Concepts

Neural Networks and Backpropagation A neural network comprises layers of interconnected neurons that process input data through weighted connections, biases, and activation functions. The backpropagation algorithm is central to training, as it calculates gradients of the loss function with respect to each weight and bias using the chain rule:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Z} \cdot \frac{\partial Z}{\partial W}$$

where L represents the loss, Z is the linear transformation $W \cdot X + b$, W denotes weights, and b signifies biases.

Softmax Activation and Cross-Entropy Loss The softmax function transforms raw output scores (logits) into probability distributions across classes:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

For classification tasks like MNIST, the cross-entropy loss measures the discrepancy between predicted probabilities and true labels:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(\hat{y}_{ij})$$

where N is the number of samples, C is the number of classes (10 for MNIST), y_{ij} is the true label, and $\hat{y}_{ij}$ is the predicted probability.

Rationale for Ten Output Neurons Given that the MNIST dataset comprises images of digits ranging from 0 to 9, a ten-class classification is essential. Each output neuron corresponds to one digit class, enabling the network to assign probabilities across these ten categories effectively.

Technologies and Methodologies

- **Eigen Library:** The Eigen library is employed for efficient matrix and vector operations, which are integral to neural network computations. Utilizing Eigen alleviates the complexity of implementing custom matrix classes, allowing for optimized performance and cleaner code.
- **C++ Object-Oriented Programming (OOP):** The project leverages OOP principles to design modular and reusable classes representing different components of the neural network, such as layers, activation functions, and the overall MLP structure.

Class Structure and Functionalities

Layer Class The `Layer` class encapsulates the properties and behaviors of a single neural network layer. It manages the weights, biases, and activation functions. During the forward pass, it computes the linear transformation of inputs and applies the specified activation function. The class also caches inputs and outputs to facilitate the backpropagation process.

MLP Class The `MLP` class orchestrates the entire neural network. It maintains a collection of `Layer` instances, manages the forward and backward propagation across all layers, handles the training loop, and oversees weight updates using an optimizer. Additionally, it provides functionalities to save and load network weights, enabling model persistence.

Activations Class The `Activations` class handles the implementation of various activation functions and their derivatives. It ensures numerical stability, especially for the softmax function, by appropriately normalizing the inputs. This class is pivotal in introducing non-linearity into the network, allowing it to learn complex patterns.

Optimizer Class The `Optimizer` class defines the strategy for updating network weights and biases based on computed gradients. In this project, the Stochastic Gradient Descent (SGD) optimizer is implemented, which adjusts parameters by moving them in the direction that minimizes the loss function.

Losses Class The `Losses` class implements different loss functions, such as Mean Squared Error (MSE) and Cross-Entropy Loss, along with their derivatives. These loss functions quantify the discrepancy between the network's predictions and the true labels, guiding the training process through gradient computations.

Algorithmic Workflow

Forward Propagation

In the forward pass, each layer processes the input data through a linear transformation followed by an activation function:

$$Z = W \cdot X + b, \quad A = f(Z)$$

Here, W represents the weights, X is the input, b is the bias, and f is the activation function. This process transforms the input data as it propagates through the network layers, culminating in the output layer's probability distribution.

Backward Propagation

During backpropagation, gradients of the loss function with respect to each parameter are computed. These gradients indicate how much each weight and bias should be adjusted to minimize the loss:

$$W = W - \eta \cdot \frac{\partial L}{\partial W}$$

where η is the learning rate. The `MLP` class utilizes these gradients to update the network's parameters, refining the model's predictions over successive training iterations.

Training Process

The training process involves repeatedly performing forward and backward passes over the dataset for a specified number of epochs. Mini-batch gradient descent is employed, where the dataset is divided into smaller batches to balance computational efficiency and convergence stability. After each epoch, the network's performance is evaluated, and weights are updated accordingly.

III. Testing

The implemented MLP was evaluated using a subset of the MNIST dataset. Due to computational constraints, a minimal neural network architecture was employed initially to verify functionality before scaling up.

Minimal Neural Network Configuration

- **Layer Sizes:** 784 (input) → 16 (hidden) → 10 (output)
- **Activation Functions:** Sigmoid for the hidden layer and Softmax for the output layer
- **Training Parameters:**
  - Epochs: 100
  - Learning Rate: 0.1
  - Batch Size: 64

Training Results

The network was trained for 100 epochs, with the loss values decreasing consistently, indicating effective learning. An example output during training:

```
Epoch 0, Loss: 2.30258
Epoch 10, Loss: 1.98765
...
Epoch 99, Loss: 0.56789
Weights saved to weights.bin
Accuracy on 1000-sample subset: 94.5%
```

Prediction Example

After training, a single image from the test set was used to evaluate the network's prediction capabilities. The prediction process involved loading the trained weights and processing the image through the network to obtain class probabilities.

```
Predicted probabilities:
0.05, 0.02, 0.85, 0.01, ..., 0.07
Sum of probabilities: 1
Predicted class: 2
```

This output displays the probability distribution across the ten classes, verifies that the probabilities sum to one, and identifies the predicted class based on the highest probability.

IV. References and Code Repositories

- **Eigen Library:** `https://eigen.tuxfamily.org/`
- **MNIST Dataset:** `http://yann.lecun.com/exdb/mnist/`
- **Statistical Learning (Gareth James, et al.):** `https://www.statlearning.com/`

- **Project Repository For my Previous MNIST solution:** https://github.com/esen-dashyam/Solutions-to-common-data-problem/blob/main/image%20classifier%20solution%20to%20problem%208.ipynb
- **Videos I followed part-1 MNIST solution:** https://www.youtube.com/watch?v=csmHZGEsONU&list=PLoROMvodv4rPP6braWoRt5UCXYZ71GZIQ&index=80
- **Videos I followed part-2 MNIST solution:** https://www.youtube.com/watch?v=VLVHpRYLaEk&list=PL1u-h-YIOL0u7R6dg_d5O5M9HUj0SFjHE
- **My favorite statistics channel STAT quest for mathematics behind neural networks** https://www.youtube.com/watch?v=IN2XmBhILt4

V. Work Distribution

As the sole contributor to this project, all tasks were undertaken individually. Below is a summary of the work distribution:

Task/Activity	**Assigned Member**	**Remarks/Notes**
Network Design and Implementation	Esen Dashnyam	Developed all classes and algorithms
Testing and Debugging	Esen Dashnyam	Conducted extensive testing on MNIST
Documentation and Reporting	Esen Dashnyam	Compiled comprehensive project report

Table 1: Work Distribution for the Project

VI. Additional Insights: Convolutional Neural Networks (CNNs)

While this project focused on implementing a fully connected neural network (MLP) for the MNIST classification task, it's noteworthy that Convolutional Neural Networks (CNNs) are often more effective for image-related tasks. CNNs utilize convolutional layers to capture spatial hierarchies in images, allowing for more efficient and accurate feature extraction.

A succinct and engaging explanation of how CNNs operate can be found in this video: *Understanding CNNs with a Simple Example*.

> The video elucidates how CNNs apply convolutional filters to input images to detect features like edges and shapes. Although our project utilized an MLP without convolutional layers, the fundamental principles of feature extraction and activation functions are shared. As demonstrated, CNNs significantly enhance classification performance by leveraging spatial relationships within image data.
>
> While the scope of this course was centered on C++ programming methodologies rather than deep learning per se, this project provided a valuable opportunity to explore the integration of programming skills with machine learning concepts. **Thank you, Professor, for facilitating this enriching experience.**

VII. Running the Project

1. Generating MNIST CSV Files

Before training the neural network, the MNIST dataset must be prepared in a suitable format. This is accomplished using a Python script that converts the dataset into CSV files. Execute the following command in your terminal:

```
python/scripts//Users/esendashnyam/Desktop/MNIST_Solution/MNIST_Data_Generator.py
```

```
python/scripts//Users/esendashnyam/Desktop/MNIST_Solution/MNIST_Data_Generator2.py
```

This script generates the following files in the `data/` directory: However data2 and data folders have pictures and data already built within them

- `train_data.csv`
- `single_picture.csv`
- `test_data.csv`
- `single_picture.csv`

2. Building the Project

1. **Navigate to the source Directory :**

   ```
   cmake -B build -S .
   cmake --build build
   cd build
   ```

2. **Configure the Project with CMake:**

   ```
   cmake ..
   ```

3. **Compile the Project:**

   ```
   make
   ```

   This process generates two executables within the `build/` directory:

   - `train`: Used for training the MLP.
   - `predict`: Utilized for making predictions on individual images.

3. Training the Model

Execute the training process with the desired network configuration. For example, to train a network with one hidden layer of 128 neurons using the sigmoid activation function, run:

```
./train --sizes 784,128,10 --activations "sigmoid,softmax" --epochs 100 --lr 0.1
```

Explanation of Arguments:

- `--sizes 784,128,10`: Defines the network architecture with an input layer of 784 neurons, one hidden layer of 128 neurons, and an output layer of 10 neurons.
- `--activations "sigmoid,softmax"`: Specifies the activation functions for each layer transition. The hidden layer uses the sigmoid activation, and the output layer uses softmax.
- `--epochs 100`: Sets the number of training epochs to 100.
- `--lr 0.1`: Establishes the learning rate at 0.1.

4. Making a Prediction

After training, use the `predict` executable to classify a single image from the test set. Follow these steps:

1. **Generate a Single Image:**

   Execute the Python script to select a random images from the MNIST test set:(inside the for loop we can determine how many pictures to by changing the number of for loops inside it.

   ```
   python scripts/MNIST_data_generator_2_tensorflow.py
   ```

   This script performs the following actions:

   - Selects a random image from the MNIST test set.
   - Saves the image as `data/single_image.csv`.
   - Outputs the true label of the selected image for comparison.

2. **Run the Prediction Executable:**

   Use the `predict` executable to classify the single image:

   ```
   ./predict
   ```

   Expected Output:

```
Predicted probabilities:
0.05, 0.02, 0.85, 0.01, ..., 0.07
Sum of probabilities: 1
Predicted class: 2
```

This output displays the probability distribution across the ten classes, verifies that the probabilities sum to one, and identifies the predicted class based on the highest probability.

VIII. Conclusion

This project successfully implements a Multi-Layer Perceptron (MLP) in C++ using object-oriented programming principles. By leveraging the Eigen library for efficient matrix operations, the neural network was able to train on the MNIST dataset and achieve commendable accuracy. Building the network from scratch provided deep insights into the mechanics of neural networks and reinforced programming skills in C++.

Future Enhancements

Future improvements could include:

- **Advanced Optimizers:** Incorporating optimizers like Adam or RMSProp for faster convergence and improved performance.
- **Regularization Techniques:** Implementing dropout or L2 regularization to prevent overfitting and enhance generalization.
- **Expanded Network Architecture:** Adding more hidden layers or experimenting with different activation functions to explore their impact on performance.
- **Unit Testing:** Developing unit tests for individual components to ensure robustness and facilitate debugging.

Acknowledgments

Thank you to Professor for guidance and support throughout this project.