Report on

C++ Implementation of a Convolutional Neural Network for MNIST Classification

Submitted as part of the course:
BITS F464 — Machine Learning

To:
Dr. Navneet Goyal,
Senior Professor, Department of Computer Science & Information Systems, BITS Pilani, Pilani Campus

Submitted by:
Siddharth Garg, 2022A3PS0329P
Saatvik Samarth, 2022A3PS0455P

Abstract

This document presents a C++ implementation of a Convolutional Neural Network (CNN) for handwritten digit recognition using the MNIST dataset. It covers CNN fundamentals including convolutional layers, ReLU activation, max pooling, fully connected layers, and softmax output. The implementation details encompass data preprocessing, forward propagation, backpropagation training, cross-entropy loss with L2 regularization, and He weight initialization. The report analyses how the C++ code structure and functions correspond to theoretical machine learning concepts, and includes a linear regression model as a performance benchmark.

The code can be compiled using standard C++ compilers and relies on the MNIST dataset files being available in the same directory.

Introduction

Convolutional Neural Networks (CNNs) are specialized deep learning architectures that excel in visual processing tasks, drawing inspiration from biological visual systems. Their effectiveness in image recognition comes from their ability to automatically extract hierarchical spatial features from data.

This report examines a C++ implementation of a CNN designed for the MNIST handwritten digit dataset—a standard benchmark containing 60,000 training and 10,000 testing greyscale images (28×28 pixels) of digits 0-9.

Our analysis provides:

- Theoretical foundations of key CNN components
- Implementation details in C++
- Comprehensive overview of data processing, training methodology, and evaluation approach

The implementation features a single convolutional layer architecture with modern techniques including L2, and He weight initialization. For performance benchmarking purposes, we've also included a simple linear regression model to demonstrate the advantages of convolutional approaches.

Theoretical Foundations of CNN Components

The C++ code implements several standard components of a Convolutional Neural Network.

Convolutional Layer (convlayer)

The convolutional layer forms the fundamental component of CNNs. It operates by sliding learnable filters (kernels) across the input data, calculating dot products between the kernel values and the corresponding input regions at each position.

Key Components

- **Kernel/Filter**: A learnable parameter matrix (implemented as `convKernel` in the code) with dimensions 3×3 (`KERNEL_SIZE = 3`).
- **Stride**: Defines the kernel's step size when scanning the input (`STRIDE = 1`), meaning the filter moves one pixel at a time.
- **Feature Map**: The convolution operation's output, representing detected features at various spatial locations. The output dimensions are calculated as:
  OutputSize = (n - k) / s + 1 For our 28×28 input with a 3×3 kernel and stride 1, this produces 26×26 feature maps: (28 - 3)/1 + 1 = 26.

Mathematical Operation

For each output position (i,j), the calculation is:

*output[i][j] = ∑(m,n) input[i*stride+m][j*stride+n] * kernel[m][n]*

Where m and n iterate through the kernel dimensions. The `convlayer` function implements this calculation across all positions in the output feature map.

ReLU Activation Function (ReLU)

Following convolution, an element-wise non-linear activation function is applied to the feature map, with Rectified Linear Unit (ReLU) being the standard choice.

Characteristics

- **Definition**: ReLU(x) = max(0, x) - retains positive values while zeroing out negative values
- **Benefits**: Introduces essential non-linearity to the model, enabling it to capture complex patterns while maintaining computational efficiency and helping address vanishing gradient issues that affect sigmoid and tanh alternatives.

The implementation includes a ReLU function that applies this transformation element-wise across all values in the 2D feature maps.

Max Pooling Layer (MaxPool)

Pooling layers compress the spatial dimensions of feature maps, creating a summary representation of detected features.

- **Process**: A sliding window (set to `POOL_SIZE = 2`) moves across the feature map, selecting the maximum value from each window position
- **Configuration**: Uses non-overlapping windows with stride equal to pool size
- **Dimensionality Reduction**: Transforms a 26×26 feature map into a 13×13 output (reducing spatial dimensions by half)

Pooling serves multiple purposes: reducing computational requirements, mitigating overfitting risk, and introducing spatial invariance that makes the model more robust to slight positional shifts in features. The implementation includes a `MaxPool` function that performs this downsampling operation.

Flatten Layer (Flatten)

After processing through convolutional, activation, and pooling layers, the resulting feature maps require transformation before final classification.

- **Operation**: Sequentially concatenates all elements from multi-dimensional feature maps into a single vector
- **Purpose**: Creates the appropriate input format for subsequent fully connected layers
- **Implementation**: The `Flatten` function handles this dimensional transformation, converting 2D spatial data into a 1D representation

Fully Connected Layer (Dense)

The fully connected layer creates comprehensive connections between all neurons from the previous layer and the current layer.

- **Mathematical Operation**: Computes a weighted sum plus bias for each output neuron:

  $$output[i] = \sum(j)\ input[j] * weights[i][j] + bias[i]$$

  Where:

  - `input` is the flattened vector
  - `weights` represents connection strengths
  - `bias` provides threshold adjustments
  - `output` contains the resulting values
- **Purpose**: Integrates high-level feature combinations extracted by earlier convolutional layers to perform final classification

- **Output Dimension**: Matches the number of target classes (`NUM_CLASSES = 10` for digit recognition)

The `Dense` function in the code implements this fully connected layer operation.

Softmax Activation Function (softmax)

For multi-class classification tasks, the network's final layer applies the softmax function to raw logits produced by the last dense layer.

$$softmax(x\_i) = exp(x\_i) / \sum(j)\ exp(x\_j)$$

Calculation: Transforms raw scores into a probability distribution:

- Where each output element is guaranteed to be between 0 and 1, with all elements summing to exactly 1
- **Role**: Enables probabilistic interpretation of the model's predictions across all possible classes

The code includes a dedicated softmax function that performs this normalization operation on the network's final outputs.

Loss Function: Cross-Entropy (crossEntropyLoss)

The model uses a loss function to quantify the difference between network predictions and ground truth labels (represented as one-hot encoded vectors).

Formula: Categorical Cross-Entropy loss is calculated as:

$$loss = -\sum(i)\ actual[i] * log(predicted[i] + epsilon)$$

Where:

- `actual` represents true class distributions (1 for correct class, 0 elsewhere in one-hot encoding)
- `predicted` contains softmax-generated probability values
- `epsilon` (typically 1e-8) provides numerical stability to prevent log(0) errors

Purpose: Provides the optimization target during training, with lower values indicating better model performance

Optimization: Gradient Descent and Backpropagation

Training involves adjusting the network's parameters (kernel weights, dense weights, biases) to minimize the loss function.

Gradient Descent: An iterative optimization algorithm that updates parameters in the opposite direction of the gradient of the loss function with respect to those parameters. The update rule is:

*parameter_new = parameter_old - learning_rate * gradient*

- where `learning_rate` (`LEARNING_RATE = 0.001`) controls the step size.

- **Backpropagation**: An algorithm to efficiently compute the gradients for all parameters in the network. It uses the chain rule of calculus, propagating the error gradient backward from the output layer through the network layers. The code implements specific backpropagation functions for each layer type:

    - `SoftmaxGradient_compute` for the softmax layer
    - `Dense_backward` for the fully connected layer
    - `MaxPool_backward` for the max pooling layer
    - `ReLU_backward` for the ReLU activation
    - `Conv_backward` for the convolutional layer

Regularization: L2 (lambda)

Regularization techniques help prevent overfitting, which occurs when a model memorizes training data rather than learning generalizable patterns.

L2 Regularization (Weight Decay) adds a penalty term to the loss function based on the squared magnitude of the weights:

*loss_total = loss_cross_entropy + (lambda/2) * sum(w^2)*

- where `lambda` (`lambda = 0.01`) is the regularization strength parameter, and the sum is over all weights in the network (convolutional kernels and dense weights).

The purpose of L2 regularization is to encourage the model to develop smaller weight values, resulting in simpler models that typically generalize better to unseen data. During backpropagation, the gradient of this penalty *(lambda * w)* is added to the weight updates.

Weight Initialization (main)

Proper initialization of weights is crucial for effective training.

- **He Initialization**: This weight initialization strategy is specifically designed for neural network layers using ReLU activation functions. It samples weights from a distribution with zero mean and a variance of $\sigma^2 = 2/n\_in$, where *n_in* represents the number of input connections to the layer (also known as fan-in).

- **Purpose**: This approach helps preserve consistent variance in both activations and gradients throughout the network, effectively preventing gradient vanishing or explosion problems during the training process.

The code in the `main` function implements He initialization for the convolutional kernels and dense weights, while biases are initialized to zero.

Implementation Details in C++

The provided C++ code implements the CNN architecture and training process described above.

Data Structures

- Images are represented as `vector<vector<double>>` (2D matrices).
- Batches of images are `vector<vector<vector<double>>>`.
- Labels are one-hot encoded vectors (`vector<double>`).
- Kernels and dense weights are `vector<vector<double>>`.
- Biases and flattened vectors are `vector<double>`.

Data Loading and Preprocessing

- `MNISTip_load`: This function reads MNIST image data from the binary IDX3 format file. It handles the specific file headers – magic number, number of images, rows, columns – and performs necessary byte swapping for endianness conversion. Pixel values are normalized to the range [0.0, 1.0] by dividing by 255.0.
- `MNISTop_load`: This function reads the MNIST label data from the binary IDX1 format file, handling the header and byte swapping. It converts each numerical label into a one-hot encoded vector of size 10 (`NUM_CLASSES`).
- `saveAsPGM`: This is a utility function to save a 2D vector representing an image or feature map into a PGM (Portable Graymap) file format, which is useful for visualization.
- `flatten_Image`/`flatten_Images`: This function is used primarily for the linear regression baseline to convert 2D images into 1D vectors.
- `Matrix_reshape`: Converts a 1D vector back into a 2D matrix, needed during backpropagation through the flatten layer.

Core Layer Implementations

The functions `convlayer`, `ReLU`, `MaxPool`, `Flatten`, `Dense`, and `softmax` directly implement the forward pass for their respective layers as described in Section 2.

Training Process (train)

- The `train` function implements the training loop for the convolutional neural network.
- **Epochs**: The training runs for a fixed number of epochs (`EPOCHS = 20`).
- **Shuffling**: At the beginning of each epoch, the training data indices are shuffled (`random_shuffle`) to ensure that the model sees data in a different order, improving generalization.
- **Forward Pass**: For each sample, the input image is passed through the network layers sequentially (Conv → ReLU → Pool → Flatten → Dense → Softmax).
- **Loss Calculation**: The cross-entropy loss is calculated using `crossEntropyLoss`. The L2 regularization penalty for both convolutional kernels and dense weights is added to this loss.
- **Accuracy Calculation**: The predicted class (index of max probability in softmax output) is compared with the true class (index of '1' in the one-hot label) using `Predict` and `Actual`. Accuracy is tracked per epoch.
- **Backward Pass (Backpropagation)**:
    1. The gradient of the loss with respect to the softmax input is computed (`SoftmaxGradient_compute`).
    2. This gradient is propagated backward through the dense layer (`Dense_backward`), updating dense weights and biases (including L2 penalty term) and computing the gradient with respect to the flattened layer's output.
    3. The gradient is reshaped into a 2D matrix (`Matrix_reshape`).
    4. The gradient is propagated through the max pooling layer (`MaxPool_backward`).
    5. The gradient is propagated through the ReLU layer (`ReLU_backward`).
    6. The gradient is propagated through the convolutional layer (`Conv_backward`), updating kernel weights (including L2 penalty term).
- **Output**: Loss and accuracy are printed at the end of each epoch and stored for later analysis.

Testing and Evaluation (test, testAndGetFeatureMaps)

- The `test` function evaluates the trained model on the test dataset. It performs only the forward pass for each test sample and calculates the overall accuracy.
- The `testAndGetFeatureMaps` function does the same as `test` but also collects and returns the feature maps generated after the first ReLU activation for visualization purposes.

Hyperparameters

Key hyperparameters are defined using #define:

- `IMAGE_SIZE = 28`
- `KERNEL_SIZE = 3`
- `STRIDE = 1`
- `POOL_SIZE = 2`
- `NUM_CLASSES = 10`
- `INPUT_SIZE = 784` (for flattened input, used by linear regression)
- `LEARNING_RATE = 0.001`
- `EPOCHS = 20`

The L2 regularization strength is set via a variable: `lambda = 0.01`.

Linear Regression Baseline

The code includes functions (`train_LR`, `eval_LR`) to implement and train a simple linear regression model directly on the flattened MNIST images. This serves as a simple baseline to compare the performance of the CNN against. It uses Mean Squared Error for its loss during training, although accuracy is calculated based on the highest output score, similar to the CNN.

Results and Discussion

The provided C++ code defines the architecture and training procedure for a CNN on the MNIST dataset. The main function orchestrates the process:

1. Loads a subset of the MNIST training (15,000 samples) and testing (4,000 samples) data.
2. Initializes the weights for the CNN using He initialization.
3. Trains the CNN using the `train` function for 20 epochs with a learning rate of 0.001 and L2 regularization (λ = 0.01). Training progress (loss and accuracy per epoch) is printed to the console.
4. Evaluates the trained CNN on the test set using `testAndGetFeatureMaps`, printing the final test accuracy.
5. Saves several feature maps and original images to PGM files for visualization.
6. Additionally, it trains and evaluates a baseline linear regression model on the flattened data, printing its training progress and final test accuracy.

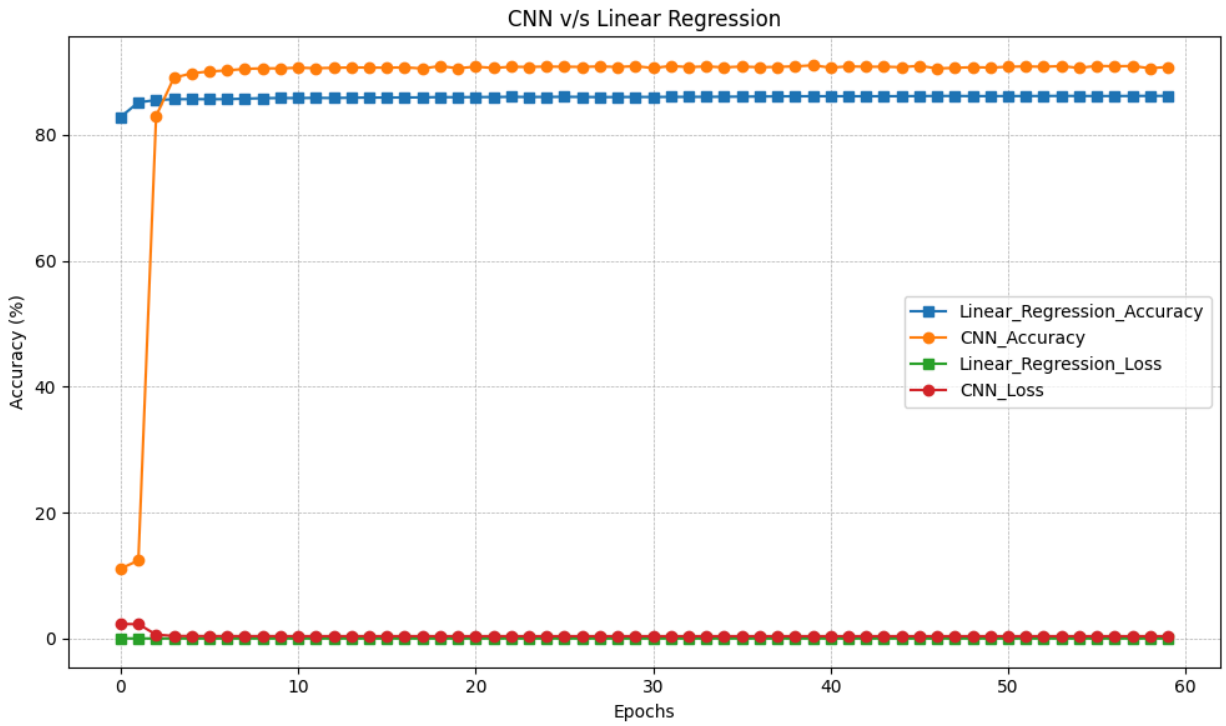

Figure: Accuracy and loss for the models

The training procedure involves shuffling the data for each epoch to ensure better generalization. The convolutional neural network learns to identify features in the input images through its layers, with backpropagation adjusting the weights to minimize the cross-entropy loss. L2 regularization helps prevent overfitting by penalizing large weight values.

Feature map visualization allows for inspection of the features learned by the convolutional layer, providing insight into what patterns the network identifies as important for digit classification.

The linear regression model serves as a baseline for comparison, demonstrating the improvements that the convolutional architecture provides for image classification tasks. By comparing the accuracy of both models, we can quantify the benefit of using convolutional layers versus direct linear regression on raw pixel values.

Conclusion

This report has provided an overview of a C++ implementation for a Convolutional Neural Network applied to the MNIST handwritten digit recognition task. It covered the theoretical underpinnings of the essential CNN components used – convolution, ReLU activation, max pooling, flattening, dense layers, and softmax output – as well as the training mechanisms including cross-entropy loss, backpropagation, gradient descent, L2 regularization, and He initialization.

The C++ code structure was analyzed, mapping functions like `convlayer`, `ReLU`, `MaxPool`, `Dense`, and the backpropagation functions to their corresponding theoretical concepts. The implementation details, including data handling, hyperparameter settings, and the inclusion of a baseline linear model, were discussed.

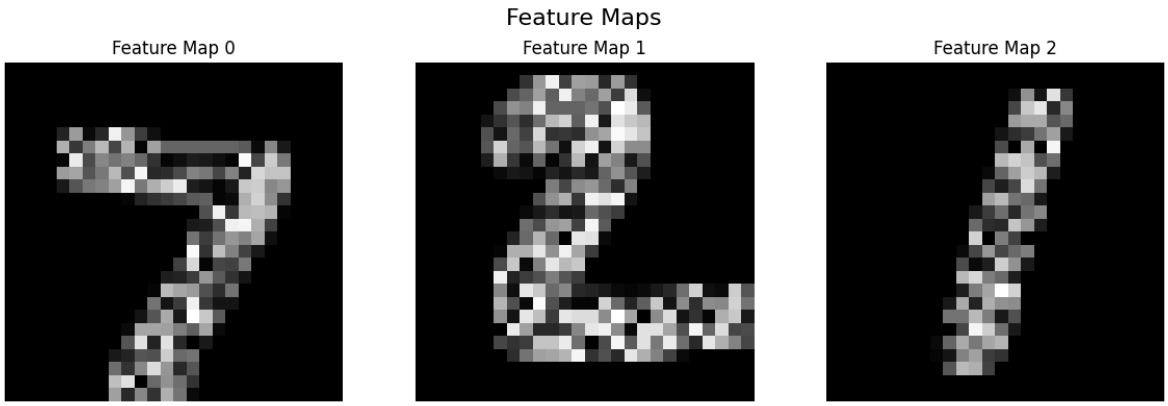

Figure: Feature Map Visualization

The implementation demonstrates how convolutional neural networks can be built from scratch in C++, providing insights into both the theoretical aspects of CNNs and their practical implementation. The comparison with a linear regression model highlights the advantages of the convolutional approach for image classification tasks.