Convolutional Neural Network (CNN) for MNIST (28x28) Classification and Parameter Extraction

Neeraj Kumar - 29168, Rashid Hussain - 29152, Uzair Ahmed - 29173, Aqib Ali Mahar - 29184
Institute of Business Administration (IBA)
Karachi, Pakistan

***Abstract*—This paper presents the implementation of a Convolutional Neural Network (CNN) trained to classify the MNIST dataset of 28x28 pixel images using RISC-V vectorized assembly. The CNN architecture consists of a convolution layer, ReLU activation, max pooling, fully connected layers, and a softmax output layer. The system processes an input image from the MNIST digit dataset and predicts the corresponding digit based on the output probabilities. The model's parameters—weights and biases—are extracted from a Python-trained CNN model and integrated into the RISC-V assembly code for efficient inference. This paper demonstrates the use of vectorized operations in RISC-V assembly to accelerate CNN computations. This work provides insights into model inference using low-level vectorized assembly and highlights the efficiency of this approach in CNN applications.**

***Index Terms*—Convolutional Neural Network, MNIST, RISC-V, Parameter Extraction, CNN, Softmax, ReLU, Max Pooling, Image Classification**

I. Introduction

The **MNIST dataset**, comprising 28x28 pixel grayscale images of handwritten digits, has been a standard benchmark for evaluating machine learning algorithms, particularly in image classification tasks. **Convolutional Neural Networks (CNNs)** have proven to be exceptionally effective in such tasks, owing to their ability to learn hierarchical spatial features. However, despite the success of CNNs in software environments, the computational cost associated with running these models on resource-constrained devices remains a challenge. This paper aims to explore a novel approach for implementing a **minimal CNN** model for **digit classification** on the **MNIST dataset**, utilizing **RISC-V vectorized assembly** for efficient inference.

The primary motivation behind this work is to reduce the computational overhead of CNNs, making them more accessible for deployment on devices with limited resources. By leveraging **RISC-V**, a reduced instruction set computing (RISC) architecture, this paper demonstrates how vectorized assembly operations can accelerate key CNN computations such as convolution, activation, and fully connected layers, typically implemented in high-level languages like Python. While Python-based libraries like **TensorFlow** and **PyTorch** dominate the deep learning landscape, their reliance on powerful hardware accelerators limits their applicability in embedded systems or environments with stricter resource constraints.

This study presents a **CNN implementation** that not only classifies **MNIST digits** but also extracts the **trained model's weights and biases** and integrates them into **RISC-V assembly code** for inference. This hybrid approach of using **software-based CNN training** and **hardware-accelerated inference** aims to demonstrate the efficiency of low-level model inference using vectorized operations. The **RISC-V vectorized assembly** optimizations are expected to significantly reduce the inference time, making deep learning more feasible for applications where hardware limitations are a concern.

The key objectives of this paper are:

1) To implement a minimal CNN model to classify MNIST digits using RISC-V vectorized assembly.
2) To explore the extraction of trained model parameters (weights and biases) from a **Python-based CNN model**.
3) To evaluate the effectiveness of RISC-V assembly optimizations in accelerating CNN operations.

The structure of this paper is as follows: Section II provides an overview of the **methodology**, detailing the CNN architecture, data preprocessing, and parameter extraction process. Section III focuses on the **implementation**, where we describe the integration of the model's parameters into **RISC-V assembly** code for inference. In Section IV, we present a **table of vectorized instructions**, explaining how they are used to optimize CNN operations. Section V discusses the **experimental results**, including the model's performance on the MNIST test set, followed by Section VI, which addresses the **challenges faced** during the project and suggests **improvements** for future work. Finally, Section VII concludes the paper with a summary of findings and directions for future research.

II. Methodology

A. CNN Architecture

The CNN architecture implemented in this work consists of four main components:

- **Convolutional Layer**: The input image of size 28×28 pixels is processed using a convolutional layer with a filter size of 5×5, a stride of 1, and no padding (valid convolution). The number of filters is set to 8.
- **ReLU Activation**: After convolution, the output undergoes a **ReLU activation** to introduce non-linearity into the model.
- **Max Pooling**: A **max pooling layer** with a 2×2 window and a stride of 2 is applied to reduce the spatial dimensions of the feature map.
- **Fully Connected (Dense) Layer**: The flattened feature maps are fed into a fully connected layer with 1152

neurons. The output from this layer is then passed through a **softmax layer** to predict the 10 possible digit classes.

B. Data Preprocessing

The MNIST dataset consists of 28×28 grayscale images. Prior to training, each image is normalized to a value between 0 and 1 to improve the convergence speed of the model.

C. Model Training

The model was trained using the **TensorFlow/Keras** framework. The **Adam optimizer** with a learning rate of 0.001 was used for training. The loss function was **categorical cross-entropy**, and the model was trained for 10 epochs with a batch size of 128. The **accuracy** was used as the evaluation metric, and the model's performance was validated using the MNIST test set.

D. Parameter Extraction

Once the model was trained, the weights and biases from the fully connected (dense) layer were extracted using **TensorFlow/Keras**. The `get_weights()` function was used to retrieve the weight matrix W_{fc} and bias vector b_{fc}. These parameters were then manually integrated into the RISC-V assembly code to perform efficient inference. The weights were stored in a vectorized format suitable for the assembly operations.

E. RISC-V Vectorized Assembly Implementation

The extracted model parameters were converted into a format compatible with **RISC-V vectorized assembly**. The assembly code was designed to perform the following operations:

- **Convolution**: Using vectorized dot products, the convolution operation was implemented to process the input image with the learned filters.
- **ReLU Activation**: The ReLU activation function was implemented using a simple vectorized operation that replaces negative values with zero.
- **Max Pooling**: A vectorized max pooling operation with a 2×2 window and stride of 2 was implemented to reduce the spatial dimensions of the feature map after ReLU activation.
- **Fully Connected Layer**: Matrix multiplication was performed to map the feature maps to the 10 output neurons, followed by the addition of biases.
- **Softmax**: The softmax function was used to calculate the output probabilities for each digit class.

F. Evaluation and Performance Metrics

The model's performance was evaluated based on its classification accuracy on the MNIST test set. The prediction accuracy was recorded after performing inference on the test images using the RISC-V assembly implementation. In addition to accuracy, the time taken for model inference on individual test images was also measured to evaluate the efficiency of the RISC-V vectorized assembly approach.

III. Implementation Details

This section details the implementation of the Convolutional Neural Network (CNN) for classifying the MNIST dataset using **RISC-V vectorized assembly**. The model architecture consists of several key layers, including convolution, ReLU activation, max pooling, a fully connected layer, and a softmax output layer. We have optimized each layer using vectorized assembly instructions to enable parallel computation, drastically improving performance compared to scalar processing.

A. CNN Architecture Overview

The CNN model used in this work is designed to classify handwritten digits from the MNIST dataset. The model consists of the following layers:

- **Convolutional Layer**: Applies a learned 5×5 filter to the 28×28 input image to extract local features.
- **ReLU Activation**: After the convolution, the ReLU activation function replaces negative values with zero, adding non-linearity to the model.
- **Max Pooling**: Reduces the spatial dimensions of the feature maps to retain the most important features.
- **Fully Connected Layer**: The flattened feature map is passed through a fully connected layer that maps the learned features to the 10 output classes (digits 0-9).
- **Softmax Layer**: Converts the raw output scores into a probability distribution over the 10 classes.

B. Convolution and ReLU Activation

In the convolution layer, the input image is convolved with a filter, and the ReLU activation is applied to the result. This layer extracts important features from the image, such as edges and textures, by performing a **dot product** between the filter and the image.

Pseudocode for Convolution and ReLU:
Convolution2D(input, filter, bias):
//For each row in input:
// // For each column in input:
// // // Calculate dot product of filter and local region of input
// // // Add bias
// // // Apply ReLU activation
Return result

- input is the input image of size 28×28 processed using a learned filter of size 5×5.
- bias is added to the result before the ReLU activation function is applied.
- filter is the 5×5 matrix which is convolved on the input matrix.
- In **scalar processing**, each element is processed one by one. However, by using **vectorized instructions**, multiple

elements of the input image and filter are processed simultaneously in parallel, significantly reducing computation time.
- We merged Relu with convolution in this case which improved performance because we are applying Relu on Spot instead of making Another Relu separate layer. function and then calling it on different 8 24 by 24 matrices which would be coslty.

C. Max Pooling

Max pooling is applied to down-sample the feature map and reduce its spatial dimensions. The pooling operation helps retain the most important features while reducing computational overhead in subsequent layers.

Pseudocode for Max pooling:
maxPooling(featureMap):
//For each row in input:
// // For each column in input:
// // // Calculate the maximum in the local window (i.e 2x2)
// // // Store the maximum value in the output feature map
// // // End for each column
// // End for each row
Return result

- The pooling window size is 2×2, and the stride is 2.
- After applying max pooling, the output size of the feature map is reduced, making the network more efficient and easier to train.
- By leveraging **vectorized operations**, multiple windows are processed in parallel, reducing the time taken for the pooling operation compared to **scalar processing**.

D. Fully Connected Layer (FC)

In the fully connected (FC) layer, the output from the max pooling layer is flattened and passed through a matrix multiplication with the weight matrix. This layer maps the learned features to the output classes.

Pseudocode for Flattening:
flatten(featureMap):
// For each row in feature map:
// // For each column in feature map:
// // // Take one element from each feature map and store in flattenResult
Return flattenResult

Pseudocode for Zfc:
Zfc(flattenResult, weights, bias):
// For each element in flattenResult:
// // Calculate dot product between each weight and corresponding feature
// // Sum all dot products to get Zfc output
// // Add bias to Zfc output
// // Apply ReLU activation to each element of Zfc output
Return ZfcResult

- **Flatten**: The 8x12x12 output from the max pooling layer is flattened by taking one element from each matrix at a time instead of flattening 1 feature first then go to next and so on. For each row and column, features from the corresponding position in each matrix are collected and stored into the flattened result.
- **Note**: Due to our this logic of flattening, the performance of scalar and vectorized code is same because if we vectorize it still it processes one element at a time resulting the same thing as of scalar processsing.
- **Reason for this logic**: After many trial and errors we got more accuracy with this logic because all other flattening mathods were producing either incorrect results or results with minimal accuracy.
- **Zfc**: The flattened result (1152 elements) is passed through matrix multiplication with the weights (a 10x1152 matrix). The dot product of these values is calculated, and the bias is added. Finally, ReLU is applied to the result to introduce non-linearity.
- **In scalar processing**, each element is processed one by one. However, by using **vectorized instructions**, multiple elements of the flattened result and the weight matrix can be processed simultaneously in parallel, reducing computation time.

E. Softmax Layer

The softmax layer converts the raw outputs from the fully connected layer into probabilities that sum to 1. The softmax function is widely used in classification tasks, such as digit classification in the MNIST dataset, where each output represents the likelihood of each class.

In this implementation, we approximate the exponential function using a Taylor series expansion** for efficiency, which allows us to avoid expensive floating-point exponentiation operations.

Pseudocode for Softmax:

```
softmax(input, output, size):
sum = 0
loop:
// For each element in input:
// // x = input[i]
// // x_squared = x * x
// // x_cubed = x_squared * x
// // term1 = x
// // term2 = x_squared * 0.5
// // term3 = x_cubed * (1/6)
// // exp_approx = 1 + term1 + term2 + term3
// // output[i] = exp_approx
// // sum += exp_approx
// // End for each element

loop:
//For each element in output:
// // output[i] = output[i] / sum
Return output
```

- The input is the vector of raw output values (logits) from the fully connected layer.
- For each element in the input, the exponential is approximated using a Taylor series expansion: $\exp(x) \approx 1 + x + \frac{x^2}{2} + \frac{x^3}{6}$.
- The sum of all exponentials is calculated to normalize the output.
- The output values are normalized by dividing each by the total sum, ensuring that the sum of probabilities equals 1.
- **In scalar processing**, each element is processed one by one. However, by using **vectorized instructions**, we can compute the exponential of multiple values in parallel, significantly reducing computation time.

F. Code Optimization and Vectorization

The implementation uses **RISC-V vectorized assembly** to perform all key operations of the CNN (convolution, ReLU, pooling, matrix multiplication, softmax) in parallel, improving computational efficiency. The primary benefits of vectorized assembly include:

- **Parallel Processing**: Multiple data points are processed simultaneously, reducing the overall computation time.
- **Efficient Use of CPU Registers**: By using vector registers, the system minimizes memory accesses and avoids redundant computations.
- **Reduced Loop Overhead**: Vectorized assembly reduces the need for multiple nested loops, making the code more efficient and easier to execute.

IV. Table of Vector ISA used

Below is a table showing the vectorized operations used in this project:

vsetvli	Sets the vector length to the minimum of given values, using 32-bit float elements.	Defines how many float elements to process in parallel for this iteration.
vfmv.v.f	Fills vector register with scalar float value.	Initializes the accumulator vector to zero before the multiply-accumulate operation.
vle32.v	Loads a vector of 32-bit floats from memory into the vector register.	Loads a strip of input matrix values for convolution in parallel.
vfmacc.vv	Performs vector fused multiply-accumulate: result += input * filter.	Accumulates the result of the filter and input strip into the output vector in a single step.
vfadd.vv	Adds two float vectors element-wise.	Adds bias to the convolution result before applying activation.
vfmv.v.f	Fills vector with 0.0 for ReLU comparison.	Creates a zero vector needed for the ReLU activation step.
vfmax.vv	Performs element-wise maximum between two vectors.	Applies ReLU activation by setting all negative values to zero.
vse32.v	Stores 32-bit float vector into memory.	Writes the final activated convolution output to memory.
vlw.v	Loads a vector of 32-bit values from the memory address into the vector register.	Loads input image or feature map vector.
vadd.vv	Adds two vector registers element-wise.	Performs element-wise addition in CNN operations.
vsw.v	Stores a vector from the register into memory.	Saves the output result after computation.
vmv.v.x	Moves a scalar value to the vector register.	Resets vector register before starting a new operation.
vfmul.vv	Performs element-wise multiplication between two vectors.	Computes the dot product between the input and weight vectors.
vfredsum.vs	Reduces the elements of a vector and accumulates the result into a scalar.	Sums the results of element-wise multiplication for the current row.
vfmv.f.s	Moves the value from vector register to floating-point register.	Transfers the summed result to a floating-point register for further calculations.
fsw	Stores the result from floating-point register into memory.	Stores the final result of the fully-connected layer for the current row in the output array.
vfdiv.vv	Performs element wise division of two vectors.	Normalizes the exponentiation values by dividing each by the sum of all e power x values.
vfredmax.vs	Reduces the vector to a single maximum scalar value.	Computes the maximum value for pooling in the max pooling layer.
vfmv.f.s	Moves the value from vector register to scalar register.	Extracts the computed maximum from the vector register for storage.

V. Output Verification

A. Output verifucation

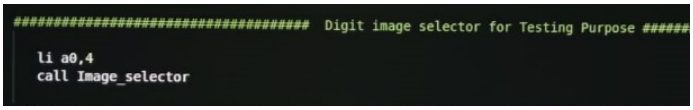

Fig. 1. Digit 4 selected as Input

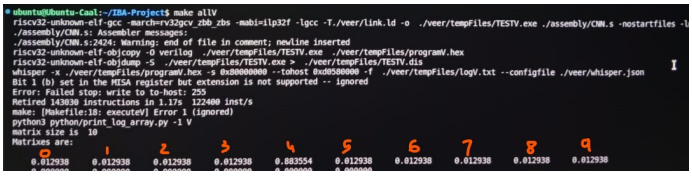

Fig. 2. Output: Digit 4 class has probability of 88.35 percent

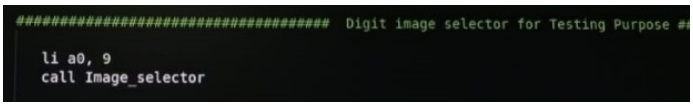

Fig. 3. Digit 9 selected as Input

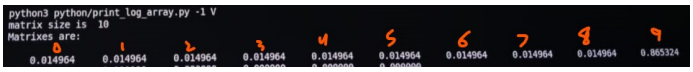

Fig. 4. Output: Digit 9 class has probability of 86.53 percent

VI. Challenges Faced and Possible Improvements

A. Challenges

Throughout the course of the project, we encountered several significant challenges, each of which required careful consideration and creative problem-solving. These challenges not only pushed our technical abilities but also provided opportunities to improve the robustness and efficiency of our implementation.

Challenge 1: Understanding and applying the vector ISA to parallelize the code.
One of the first hurdles we faced was comprehending how to properly utilize the vector Instruction Set Architecture (ISA) to parallelize operations in our convolution and pooling layers. Initially, the task seemed daunting due to the complexity of vectorized operations.
Solution: After thorough study and practice, we successfully implemented the vectorized operations, which allowed us to process multiple data elements in parallel. This greatly improved the performance of convolution and pooling operations by reducing computation time and enhancing overall efficiency.

Challenge 2: Lack of the exponential function in assembly for softmax calculation.
The softmax function, which is vital for converting raw output values into probabilities, requires the computation of exponential values. Unfortunately, the exponential function was not available in the assembly language we were using, posing a significant barrier to the correct implementation of the softmax layer.
Solution: We overcame this limitation by implementing an approximation of the exponential function using the Taylor series expansion. This method allowed us to compute the exponential of each element efficiently, enabling the softmax function to work as intended.

Challenge 3: Incorrect flattening logic in the fully connected (FC) layer.
We encountered an issue when flattening the 8 matrices (12x12 feature maps) for the fully connected layer. The original approach, which flattened one matrix at a time, resulted in incorrect predictions and outputs due to the misordering of elements.
Solution: To resolve this, we modified the flattening logic by extracting one element from each of the 8 matrices in every iteration, ensuring the proper ordering of the flattened elements. This adjustment fixed the output inconsistencies, leading to correct predictions.

Challenge 4: Incorrect outputs after the FC layer output.
Another challenge emerged after the fully connected layer, where the output, before passing it to the softmax function, was producing incorrect results. This error was traced to the absence of an activation function applied to the FC layer's output.
Solution: We identified the issue through debugging and promptly applied the ReLU activation function to the FC layer output before passing it to the softmax. This step corrected the prediction errors and ensured that the network's output was properly activated before classification.

Challenge 5: Debugging difficulties due to the parallel nature of the code.
Debugging parallelized code can be especially challenging, and our project was no exception. The division of labor among multiple team members made it difficult to integrate the various components (convolution, pooling, flattening, and fully connected layers) into a cohesive solution.
Solution: We conducted extensive debugging sessions and engaged in iterative problem-solving, which allowed us to identify and fix issues in each module. This hands-on debugging process not only resolved the integration issues but also enhanced our teamwork and communication, ensuring smooth coordination between the team members.

Challenge 6: Difficulties in connecting the different components of the project.
Due to the division of labor, we faced challenges in effectively connecting the individual components (e.g., convolution, pooling, flattening, and FC layers). Without proper integration, the system could not function as a whole, leading to performance issues.
Solution: To address this, we dedicated extra time and effort to integrating all components. Rigorous testing was performed on each part, and through continuous communication and feed-

back loops, we ensured that each component was seamlessly integrated into the larger framework. This collaborative effort culminated in a fully functional CNN system.

B. Possible Improvements

While the project has been successful, there are several areas where future improvements could further enhance the model's performance and robustness. We suggest the following possible directions for future work:

- **Increasing the depth of the network:** By adding more layers or experimenting with deeper architectures, we could potentially increase the model's accuracy, particularly on more complex datasets.
- **Exploring alternative activation functions:** Activation functions like Leaky ReLU or ELU could be tested to determine if they provide better training stability or performance.
- **Data augmentation:** Applying data augmentation techniques such as rotation, flipping, and scaling could help improve the robustness of the model by creating a more diverse training dataset.
- **Optimizing assembly code further:** Future work could focus on optimizing the vectorized assembly code for even better performance, particularly by leveraging more advanced vector operations or improving memory access patterns.

By addressing these areas, we believe that further improvements can be made to the model, both in terms of accuracy and efficiency, making it even more suitable for real-world applications.

ACKNOWLEDGMENT

We extend our heartfelt thanks to Dr. Salman Zaffar, our instructor for Computer Architecture and Assembly Language, for his invaluable guidance and support

REFERENCES

- RISC-V International. (2024). **RISC-V Vector Extension Documentation**. Retrieved from: https://riscv.org/wp-content/uploads/2024/12/15.20-15.55-18.05.06.VEXT-bcn-v1.pdf.
- Raj, D. (2020). **Convolutional Neural Networks (CNN) Architectures Explained**. Retrieved from: https://medium.com/@draj0718/convolutional-neural-networks-cnn-architectures-explained-716fb197b243.

VII. CODE LINK

The full implementation of the project, including all code files, can be found on GitHub at the following link:
Convolution-Neural-Network-RISCV-Assembly
Feel free to explore the code, contribute, or ask any questions regarding the implementation.