EE 533 Optional Lab #10 (Extra Credit)
Neural Processing Unit Hardware Accelerator for Network Processor.
Instructor: Young Cho - youngcho@isi.edu

1. Overview

In this lab, you'll design a neural processor by creating a Multiply-Accumulate (MAC) module with four Float16 multipliers and an adder to sum their outputs, processing network packet data(payloads). Payloads would be 8*8 MNIST handwritten digit images(Binary images 0/1). Compare the losses of your hardware design against a software implementation using PyTorch or TensorFlow, and document your findings in a project report. You will need to train the model separately on your local machine/cloud, load the trained weights into NetFPGA either via payload or via register interface.

2. Background

Deep neural networks (DNNs) are increasingly being used to classify network traffic payloads as malicious or benign in real time. There has been considerable research in this area, including Cisco's work on an AI-based firewall that leverages advanced machine learning techniques for improved threat detection.

One simple Deep Learning method is an Artificial Neural Network (ANN), which can be expressed by the following equation:

$$y_j = \sigma\left(\sum_{\{i=1\}}^{\{n\}} w_{\{ij\}} x_i\right)$$

Here x_i are 16-bit floating point data (in this lab, your packet payload), and w_{ij} is the weight for x_i (these weights are learned during the training process). The function σ is ReLU activation which zeros out the negative output ($\sigma(x) = \max\{0,x\}$)

3. Implementation

Step 1: Firstly, you must get familiar with how the ANN works. There are many free online resources for this. At the lowest level of computation, the following is what you will need to implement.

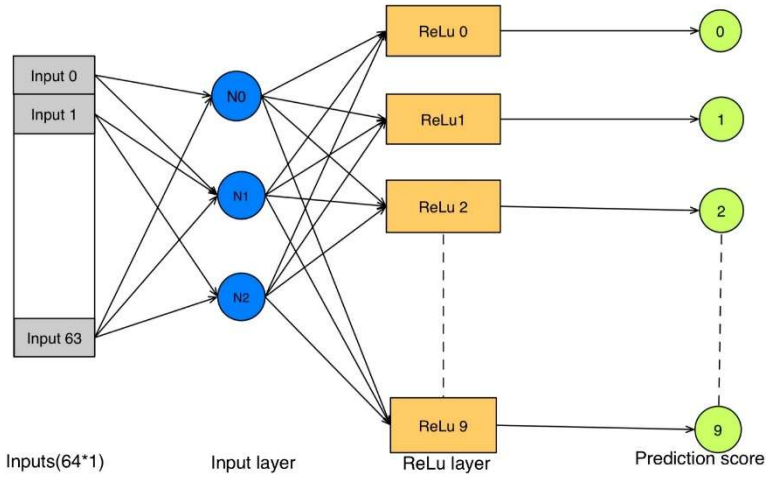

Figure 1: An Example 1-layer Artificial Neural Network

Once you have a good idea of the neural network architecture, here are some things that are worth noting about the ANN architecture you will be implementing

1. Input Processing:
    - **Data:** We use binary images (0/1), not RGB.
    - **Resizing:** MNIST images originally sized 28×28 are resized to 8×8, so each image has 64 bits.
2. First Layer (Hidden Layer):
    - **Neurons:** Contains 3 neurons.
    - **Connections:** Each of the 64 input bits is fully connected to each neuron.
    - **Weights:** Total weights = 64 inputs × 3 neurons = 192 weights.
    - **Computation:** Each neuron performs a dot product (matrix multiplication) between the 1×64 input vector and its corresponding 64×1 weight vector, outputting a single fp16 value.
3. Threaded Computation:
    - **Threads:** There are 4 threads (T0, T1, T2, T3).
    - **Parallel Processing:** Threads T1, T2, and T3 can independently handle the computations for each of the 3 neurons.
    - **Multiplier Simplification:** Since the inputs are binary (0 or 1), each multiplication effectively outputs either 0 or the weight value. This can be implemented with a simple multiplexer (MUX).
4. Output Layer:
    - **Inputs:** The 3 outputs from the hidden layer serve as inputs.
    - **Connections:** Each hidden neuron is connected to all 10 output neurons.
    - **Weights:** Total weights = 3 hidden neurons × 10 output neurons = 30 weights.
    - **Computation:** The output layer computes raw logits through another set of matrix multiplications.

Step 2: Install tools and train the model and save the weights.

You can install the tools under Linux by typing the following under Linux:

```
> pip install torch torchvision scikit-learn numpy
```

Then you can use the following Python script to train your ANN to recognize example images of digits. You should go through the code to figure out exactly what the script is doing so that you can modify it for your use later (attached Python code).

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from torch.utils.data import TensorDataset, DataLoader

import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
```

```
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt

# 1) Load data and create a parallel array of indices
digits = load_digits()
X, y = digits.images, digits.target
indices = np.arange(len(X))

# 2) Threshold data for training, flatten, etc.
X_binary = np.where(X > 8, 1.0, 0.0).reshape(-1, 64).astype(np.float32)

# 3) Split both your data and indices
X_train, X_test, y_train, y_test, idx_train, idx_test = train_test_split(
    X_binary, y, indices, test_size=0.2, random_state=42
)

# 4) Convert to tensors
X_train = torch.tensor(X_train, dtype=torch.float16)
y_train = torch.tensor(y_train, dtype=torch.long)
X_test = torch.tensor(X_test, dtype=torch.float16)
y_test = torch.tensor(y_test, dtype=torch.long)

train_loader = DataLoader(TensorDataset(X_train, y_train), batch_size=32, shuffle=True)
test_loader = DataLoader(TensorDataset(X_test, y_test), batch_size=32, shuffle=False)

# Define your minimal model with ReLU instead of sigmoid
class MinimalFNN(nn.Module):
    def __init__(self):
        super(MinimalFNN, self).__init__()
        self.hidden = nn.Linear(64, 3, bias=False)  # 3 neurons in hidden layer
        self.output = nn.Linear(3, 10, bias=False)  # 10 output classes

    def forward(self, x):
        # Replace sigmoid with ReLU
        x = torch.relu(self.hidden(x))
        x = self.output(x)
        return x

# Instantiate model
model = MinimalFNN().half()

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.5)

# Train
num_epochs = 70
for epoch in range(num_epochs):
    for inputs, labels in train_loader:
        inputs = inputs.half()
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

# Evaluate accuracy
correct, total = 0, 0
with torch.no_grad():
    for inputs, labels in test_loader:
        inputs = inputs.half()
        outputs = model(inputs)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Test Accuracy = {100 * correct / total:.2f}%")

# 5) Show 10 random test images with predictions
num_images = 10
fig, axes = plt.subplots(2, 5, figsize=(12, 6))
axes = axes.ravel()
```

```
for i in range(num_images):
    # Pick random index *within the test set*
    rand_idx = np.random.randint(0, len(X_test))

    # Map to original index
    original_idx = idx_test[rand_idx]

    # Run inference
    sample = X_test[rand_idx].view(1, -1).half()
    with torch.no_grad():
        output = model(sample)
        predicted_digit = torch.argmax(output).item()

    # Display the original 8x8 grayscale image
    axes[i].imshow(digits.images[original_idx], cmap='gray')
    axes[i].set_title(f"Pred: {predicted_digit}\nActual: {digits.target[original_idx]}")
    axes[i].axis("off")

plt.tight_layout()
plt.show()

# Save weights as float16
hidden_weights = model.hidden.weight.detach().cpu().half().numpy()
output_weights = model.output.weight.detach().cpu().half().numpy()

np.savetxt("hidden_weights.txt", hidden_weights, fmt="%.5f")
np.savetxt("output_weights.txt", output_weights, fmt="%.5f")

print("Hidden layer weights saved to hidden_weights.txt (float16 in text form).")
print("Output layer weights saved to output_weights.txt (float16 in text form).")
```

Step 3: Understand the inference logic as well as hardware and software implementation. The inference is a simple forward pass, with no loss calculation or backpropagation. Check the following CPP code (attached) to understand the forward pass.

```
#include <iostream>
#include <cmath>
#include <cstdlib>
#include <ctime>

using half = __fp16;

inline half half_relu(half x)
{
    float fx = static_cast<float>(x);
    float out_f = (fx > 0.0f) ? fx : 0.0f;
    return static_cast<half>(out_f);
}

int main()
{
    // 8*8 0/1 image will be input so 64 inputs
    // Number of weights =   number_of_hidden_neurons*number_of_input  +
number_of_output_neurons*number_of_hidden_neurons
    // Number of hidden neurons = 3*64 + 10*3 = 192


    // FOR NOW I have set the code to have random  values for weights and inputs, but you can
have a them
    // sent to netfpga via payload or hard coded to your ROM

    // SECTION -1
    //Ignore this for now, just random values initialized for weighsts and inputs
    std::srand(static_cast<unsigned int>(std::time(nullptr)));

    const int INPUT_SIZE  = 64;
    const int HIDDEN_SIZE = 3;
```

```
    const int OUTPUT_SIZE = 10;

    half input[INPUT_SIZE];
    for(int i = 0; i < INPUT_SIZE; i++) {
        // random [0,1)
        float val = static_cast<float>(std::rand()) / RAND_MAX;
        input[i] = static_cast<half>(val);
    }

    half hidden_weights[HIDDEN_SIZE][INPUT_SIZE];
    for(int h = 0; h < HIDDEN_SIZE; h++) {
        for(int j = 0; j < INPUT_SIZE; j++) {
            // random in [-1, 1]
            float val = 2.0f * (static_cast<float>(std::rand()) / RAND_MAX) - 1.0f;
            hidden_weights[h][j] = static_cast<half>(val);
        }
    }

    half output_weights[OUTPUT_SIZE][HIDDEN_SIZE];
    for(int k = 0; k < OUTPUT_SIZE; k++) {
        for(int h = 0; h < HIDDEN_SIZE; h++) {
            float val = 2.0f * (static_cast<float>(std::rand()) / RAND_MAX) - 1.0f;
            output_weights[k][h] = static_cast<half>(val);
        }
    }

    //  Number of hidden layers = 3
    //  You have 4 threads so unroll the outer loop 3 times to run each neuron per thread

    half hidden[HIDDEN_SIZE];

    //SECTION 2:
    for(int h = 0; h < HIDDEN_SIZE; h++) {
        float sum = 0.0f;
        for(int j = 0; j < INPUT_SIZE; j++) {
            sum += static_cast<float>(hidden_weights[h][j]) *
                   static_cast<float>(input[j]);
        }
        hidden[h] = half_relu(static_cast<half>(sum));
    }

    // Have a shared semmaphore across three threads to synchronize the threads(each thread
increments the semaphore)
    //Unrolling end

    // When all the three threds complete (when you read the semaphore vaue as 3) then one of the
thread T0 (lets call it a host code thread) will execute the next steps
    half outputs[OUTPUT_SIZE];
    //SECTION 3
    for(int k = 0; k < OUTPUT_SIZE; k++) {
        float sum = 0.0f;
        for(int h_ = 0; h_ < HIDDEN_SIZE; h_++) {
            sum += static_cast<float>(output_weights[k][h_]) *
                   static_cast<float>(hidden[h_]);
        }
        outputs[k] = static_cast<half>(sum); // raw logits (no activation)
    }

    std::cout << "Output logits (printed as floats, stored in half):\n";
    for(int k = 0; k < OUTPUT_SIZE; k++) {
        // cast to float just for printing
        float val = static_cast<float>(outputs[k]);
        std::cout << val << "  ";
    }
    std::cout << "\n";

    int best_class = 0;
    float best_val = static_cast<float>(outputs[0]);
    for(int k = 1; k < OUTPUT_SIZE; k++) {
        float val = static_cast<float>(outputs[k]);
        if(val > best_val) {
```

```
            best_val = val;
            best_class = k;
        }
    }
    std::cout << "Predicted class: " << best_class << "\n";

    return 0;
}
```

There are three sections mentioned in the comments:

Section 1: Is fetching trained weights (192 + 30 weights) and input(64)

In this part of the code, I have just taken random values for now, but for the implementation, you should send the 64-bit input data as payload to your neural processor. The weights can be hardcoded to a ROM that is common to all threads.

Section 2: This is where the computation of all neurons happens.

The outer loop is iterating over 1st layer neurons 3 times. Each neuron is a matrix multiplication of [64x1] * [1*64].

```
//SECTION 2:
for(int h = 0; h < HIDDEN_SIZE; h++) {
    float sum = 0.0f;
    for(int j = 0; j < INPUT_SIZE; j++) {
        sum += static_cast<float>(hidden_weights[h][j]) *
               static_cast<float>(input[j]);
    }
    hidden[h] = half_relu(static_cast<half>(sum));
}

// Have a shared semmaphore across three threads to synchronize the threads(each thread increments the semaphore)
//Unrolling end
```

You can make three independent programs for T1, T2, and T3 that essentially does the same thing parallelly. Each thread needs to have a fp16 multiplier (essentially a MUX that chooses a value between 0 and Weight(w)).

You should build simple hardware using Verilog for ReLU.

Section 3: When threads T1, T2 and T3 complete the thread T0 can execute the remaining section of the code.

You can send the inputs (x_i) and weights (w_i) as payload and compute the MAC (Multiply-Accumulate) outputs for your packets.

4. Integration to the Network Processor

Architect and develop your Neural Processing Unit. You are free to architect it any way you believe is the best way. Options range from creating bf16 and ReLU SIMD or SISD instructions, to dedicated hardware acceleration module. Once you have the design, integrate it with your network processor with its software interface.

Prepare an experiment with input images embedded in the network packets and your ARM code for your network processor that recognizes the images on the fly.

5. Scope and submission guidelines of this lab

- Generate and tabulate the outputs from your MAC (Multiply-Accumulate) hardware for sample inputs. Compare these results against a golden reference solution computed using libraries like PyTorch or TensorFlow. Finally, report the error regarding **Mean Squared Error (MSE)**.
- Submit all your schematic files and source codes to GitHub, ensuring the commit history is properly logged.