SEAI_2024_R12

Giulio Capecchi, Jacopo Niccolai

December 2024

1 Introduction

This project focuses on the synthesis of the forward pass for three types of neural network architectures: a **Multilayer Perceptron (MLP)**, a **Convolutional Neural Network (ConvNet)**, and a **Transformer**, implemented on an FPGA. To achieve this, the network parameters were first obtained using Python and the *PyTorch* library. These parameters were subsequently hardcoded into `C` code, enabling the hardware synthesis process.

2 Project Description

2.1 Workflow Overview

The neural networks were constructed and trained using the *PyTorch* library. Once trained, the weights and biases were exported to be hardcoded into the corresponding `C` implementation. The `C` code was specifically designed to be compatible with FPGA synthesis tools, such as Vitis HLS, ensuring efficient hardware synthesis.

2.2 C Implementation

The `C` code developed includes the forward pass for:

- **MLP**: implementation of propagation through dense layers.
- **ConvNet**: handling of convolution and pooling operations.
- **Transformer**: managing complex operations like attention.

HLS INLINE forza l'inserimento completo del corpo di una funzione o di un ciclo direttamente nel punto in cui è chiamato, eliminando l'overhead associato alla chiamata di funzione o all'allocazione di risorse hardware separate. Riduce la latenza perché elimina il ritardo di chiamata della funzione. Può aumentare l'area hardware utilizzata, perché la logica della funzione viene replicata ovunque venga chiamata. Tipicamente per funzioni semplici o molto usate, per ridurre la latenza.

HLS PIPELINE La direttiva HLS PIPELINE suddivide un ciclo o una funzione in più fasi (pipeline), permettendo l'esecuzione di più iterazioni o operazioni simultaneamente, aumentando il throughput del design. Permette l'elaborazione di nuove iterazioni a ogni ciclo di clock (o a intervalli specificati, chiamati Initiation Interval - II). Aumenta la velocità del sistema a scapito del consumo di risorse hardware. Opzioni: II=N (specifica l'intervallo di avvio delle iterazioni, ad esempio, 1 ciclo di clock tra una e l'altra). rewind (riavvia automaticamente il ciclo alla fine). Uso tipico Nei loop che processano grandi quantità di dati. Per aumentare il throughput in operazioni ripetitive.
The network parameters (weights and biases) were directly integrated into the code in a hardcoded manner.

3 Code Architecture

3.1 C File Structure

The forward pass is implemented using a sequence of functions for each layer type:

- Activation functions (`relu`, `softmax`, etc.).
- Functions for convolution and pooling operations.
- Functions for attention mechanisms in Transformers.

3.2 Hardware Synthesis

The code was designed to be compatible with tools such as Vitis HLS, leveraging specific pragmas to optimize the implementation.

4 Multi-Layer Perceptron

Let's analyze the implementation of the forward pass for a Multi-Layer Perceptron. The forward pass for an MLP consists of propagating the input through a series of dense layers, each followed by an activation function. The code for it can be found inside the `PyTorch` folder, that contains the notebooks used to train the models and export their parameters.

4.1 Dataset

The MLP was trained using the well-known *Iris* dataset, which contains 150 samples of iris flowers, each with four features and a class label (the last value of each row). There is a total of three classes: *setosa*, *versicolor*, and *virginica*. The dataset was split into training and test sets, with 80% of the samples used for training and 20% for testing.
An example of the dataset is shown below:

```
sepal_length,sepal_width,petal_length,petal_width,species
5.1,3.5,1.4,0.2,setosa
5.7,2.8,4.5,1.3,versicolor
6.1,2.6,5.6,1.4,virginica
...
```

4.2 PyTorch Model

4.2.1 Model Architecture

The architecture of the MLP model consists of three fully connected (dense) layers. The input layer has 4 neurons corresponding to the 4 features of the Iris dataset. The first and second hidden layers each have 10 neurons, and the output layer has 3 neurons corresponding to the 3 classes of the Iris dataset. The chosen activation function is the ReLU function, which is applied after each dense layer except the output layer. The forward pass of the model involves applying the ReLU activation function after the first and second layers.
ReLU is defined as:

$$\text{ReLU}(x) = \max(0, x)$$

The model was defined as follows, using the *PyTorch* library:

```
# Define the MLP model
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(4, 10)
        self.fc2 = nn.Linear(10, 10)
        self.fc3 = nn.Linear(10, 3)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

4.2.2 Model Training

For the training phase, we first defined the model, loss function, and optimizer. We utilized the *CrossEntropyLoss* loss function, which is commonly used for multi-class classification problems, and the *Adam* optimizer, which is an adaptive learning rate optimization algorithm.

```
model = MLP()

# Check if GPU is available
```

```
device = torch.device('cuda' if torch.cuda.is_available()
    else 'cpu')
model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

The following code snippet demonstrates the trainign process using *PyTorch*. The model is trained for 100 epochs.

```
# Training loop
NUM_EPOCHS = 100
for epoch in range(NUM_EPOCHS):
    model.train()
    running_loss = 0.0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device
            )

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    # Evaluate the model after each epoch
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(
                device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1) # Get
                the class index with the highest probability
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = correct / total

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{NUM_EPOCHS}], Loss: {
            running_loss/len(train_loader):.4f}, Accuracy: {
            accuracy * 100:.2f}%')
```

The results of the training process are reported in the table below:

Epoch	Loss	Train Accuracy	Test Accuracy
10	0.3258	90.48%	88.89%
20	0.1060	97.14%	97.78%
30	0.1312	95.24%	97.78%
40	0.0853	97.14%	100.00%
50	0.0675	98.10%	100.00%
60	0.0971	96.19%	97.78%
70	0.0952	96.19%	97.78%
80	0.1020	96.19%	97.78%
90	0.0929	96.19%	97.78%
100	0.0788	94.29%	97.78%

Table 1: Training loss, train accuracy, and test accuracy of the MLP over 100 epochs.

The training process shows that the model is able to achieve a high level of accuracy on both the training and test sets: this is to be expected given the simplicity of the Iris dataset and the effectiveness of the MLP architecture in solving such problems. Given the relatively small dataset and the fact that the model achieves near-perfect accuracy on the test set, we can conclude that the model is generalizing well.

4.2.3 Exporting Parameters

The trained model's parameters were exported to be hardcoded into the `C` implementation. The weights and biases of each layer were extracted and saved in a txt file, *weights.txt*, as shown below:

```python
import numpy as np

weights = {}
for name, param in model.named_parameters():
    weights[name] = param.detach().numpy()

# Print their shapes to verify the network architecture
for name, weight in weights.items():
    print(f"{name}: {weight.shape}")

with open('./weights.txt', 'w') as f:
for name, weight in weights.items():
    f.write(f"{name}\n")
    np.savetxt(f, weight, fmt='%f')
```

4.3 C Implementation

4.3.1 Implementation

The forward pass for an MLP is implemented in `C` using a sequence of functions for each layer type. The function takes four input features and processes them through three layers, each with its own weights and biases. The output is the predicted class index.

4.3.2 Forward Pass

The forward pass function is defined as follows:

```
int forward(float input0, float input1, float input2, float
    input3) {
    const int input_sizes[4] = {4, 10, 10, 3};
    const int num_layers = 3;

    float current_input[MAX_NEURONS];
    float next_input[MAX_NEURONS];

    current_input[0] = input0;
    current_input[1] = input1;
    current_input[2] = input2;
    current_input[3] = input3;

    for (int i = 0; i < num_layers; i++) {
        #pragma HLS UNROLL
        Layer *layer = &mlp.layers[i];
        for (int j = 0; j < input_sizes[i + 1]; j++) {
            float sum = layer->biases[j];

            for (int k = 0; k < input_sizes[i]; k++) {
                sum += layer->weights[j][k] * current_input[
                    k];
            }
            next_input[j] = reLu(sum);
        }

        for (int j = 0; j < input_sizes[i + 1]; j++) {
            current_input[j] = next_input[j];
        }
    }

    int max_index = 0;
    float max = current_input[0];
    for (int i = 1; i < NUM_CLASSES; i++) {
        #pragma HLS UNROLL
        if (current_input[i] > max) {
            max = current_input[i];
```

```
36              max_index = i;
37          }
38      }
39      return max_index;
40  }
```

5 Results

TODO

6 Conclusions

TODO