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.

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.

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 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
```

4.3 PyTorch Training

The following code snippet demonstrates the trainign process using *PyTorch*. The model is trained for 100 epochs, and the loss and accuracy are printed every 10 epochs.

```
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
import numpy as np

# Load and preprocess the dataset
data = np.loadtxt('../datasets/iris_dataset.txt')
X = data[:, :4] # get the first 4 columns (features)
y = data[:, 4] # get the last column (labels)

# Encode labels
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y) # fit_transform returns
    the encoded labels as integers [e.g. setosa -> 0]

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=42)

# Convert to PyTorch tensors
```

```
X_train = torch.tensor(X_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.long)
y_test = torch.tensor(y_test, dtype=torch.long)

# 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

# Initialize the model, loss function, and optimizer
model = MLP()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Training loop
num_epochs = 100
for epoch in range(num_epochs):
    model.train()
    optimizer.zero_grad()
    outputs = model(X_train)
    loss = criterion(outputs, y_train)
    loss.backward()
    optimizer.step()

    # Evaluate the model
    model.eval()
    with torch.no_grad():
        outputs = model(X_test)
        _, predicted = torch.max(outputs.data, 1)
        accuracy = (predicted == y_test).sum().item() /
            y_test.size(0)

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

Results of the training process are shown below:

```
Epoch [10/100], Loss: 1.0095, Accuracy: 36.67%
Epoch [20/100], Loss: 0.8086, Accuracy: 70.00%
Epoch [30/100], Loss: 0.5072, Accuracy: 80.00%
Epoch [40/100], Loss: 0.3274, Accuracy: 100.00%
Epoch [50/100], Loss: 0.1901, Accuracy: 100.00%
Epoch [60/100], Loss: 0.1097, Accuracy: 100.00%
Epoch [70/100], Loss: 0.0783, Accuracy: 100.00%
Epoch [80/100], Loss: 0.0665, Accuracy: 100.00%
Epoch [90/100], Loss: 0.0614, Accuracy: 100.00%
Epoch [100/100], Loss: 0.0586, Accuracy: 100.00%
```

Figure 1: Training loss and accuracy of the MLP over 100 epochs.

Accuracy on the test set reached 100% after 100 epochs of training.

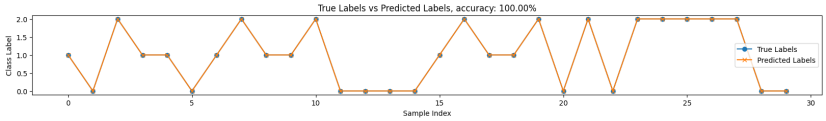

Figure 2: Accuracy of the MLP on the test set after 100 epochs of training.

4.4 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 format compatible with the hardware implementation.

```python
# Export the trained model's parameters
weights = {name: param.detach().numpy() for name, param in
    model.named_parameters()}

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

4.5 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.
The forward pass function is defined as follows:

```c
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];
            max_index = i;
        }
    }
    return max_index;
}
```

5 Results

TODO

6 Conclusions

TODO