MoonRiver: Deep Neural Network in C++

Chung-Yi Weng
Computer Science & Engineering
University of Washington
chungyi@cs.washington.edu

Abstract

Artificial intelligence resurges with its dramatic improvement in recognition and prediction. The driving force is deep neural network. Although there are lots of existed packages, like Caffe, TensorFlow, PyTorch, or MXNet, to help people apply neural network technique to the problems, the running algorithm behind them is obscure. In order to overcome this, we decide to implement deep neural network in C++ from scratch, called MoonRiver. This is not only for uncovering the secrets behind deep neural network but also offering a good code base to help us discover more essential advancement in the future.

There are several goals in developing MoonRiver. The first is independence: we hope MoonRiver could be lightweight. It shouldn't depend on any third-party libraries, and therefore could be easily compiled just using standard C++ compiler. It makes it possible to port MoonRiver on any OSes and machines. The second is scalability: MoonRiver should make users easily design any networks and scale to large ones with minimum effort.

We demonstrate the effectiveness of MoonRiver by training and testing auto-encoder and LeNet. The codes used to define these networks are concise and the experimental results are promising.

1 Overview

Deep neural network is a black box. The packages of neural network, like Caffe, TensorFlow, PyTorch, or MXNet, are another black boxes. In order to uncover the secrets behind these boxes, we want to implement deep neural network in C++ from scratch, called MoonRiver. This is not only for comprehensively understanding the algorithm running in deep neural network but also for offering a good code base to improve quality and performance of the learning technique essentially in the future.

Several properties are desired in designing MoonRiver.

1. **Independence:** MoonRiver is lightweight. It shouldn't have any dependencies on any third-party libraries. It could be easily compiled by standard C++ compiler.
2. **Portability:** MoonRiver should be easily ported on any OSes and machines.
3. **Convenience:** MoonRiver makes it easy for users to design any networks they want.
4. **Scalability:** MoonRiver should be easily scaled to large network with minimum effort.

The report is organized as follows. We describe MoonRiver supported feature in Sec. 2. Sec. 3 introduces how to use MoonRiver to design and train a customized network, taking LeNet as a running example. In Sec. 4, we demonstrate the effectiveness of MoonRiver by showing the training setting and testing results on two networks - auto-encoder for image generation and LeNet for image classification. The possible extensions and future works are explained in Sec. 5

2 Supported Features

Here we describe all currently MoonRiver supported features. In a nutshell, MoonRiver implements most necessary layers and activation functions to do image classification and generation, and also offers several effective optimizers to update training coefficients.

Here is the summary of all features MoonRiver supports:

1. **Layer**
   (a) Convolutional Layer
   (b) Fully Connected Layer
   (c) Max Pooling Layer
   (d) Softmax Layer
   (e) Flatten Layer
2. **Activation:**
   (a) Linear
   (b) Relu
   (c) Tanh
   (d) Sigmoid
3. **Optimizer:**
   (a) SGD
   (b) Momentum
   (c) RMSprop
   (d) Adam
4. **Cost Function:**
   (a) Mean Square Error
   (b) Negative Log Likelihood
5. **Misc:**
   (a) MNIST Data Loader
   (b) Mini-batch Random Sampler
   (c) Network Saving and Loading

Notice DropOut and Batch Normalization are not supported right now, should be added on in the future.

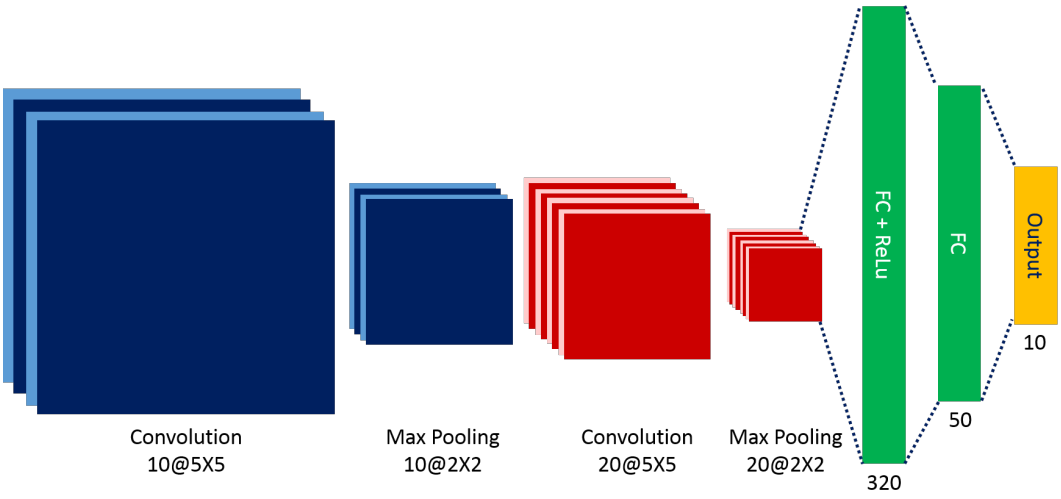

Figure 1: The network architecture of LeNet.

3 Usage

Here we explain how to use MoonRiver to design a customized network and how to train it. We take LeNet as a running example because of its elegant architecture and prestigious fame on image classification.

Although I believe most readers have already known the architecture of LeNet, for completeness of the report, I still illustrate this famous architecture in Figure. 1

If we use MoonRiver to implement LeNet, the codes we have to write are only as follows:

```
void LeNet::Init()
{
    int total_layer = 8;
    m_layers = std::vector<Layer*>(total_layer);

    m_layers[0] = new ConvolutionalLayer(1, 10, 5, RELU);
    m_layers[1] = new MaxPoolLayer(2);
    m_layers[2] = new ConvolutionalLayer(10, 20, 5, RELU);
    m_layers[3] = new MaxPoolLayer(2);

    m_layers[4] = new FlattenLayer();

    m_layers[5] = new FullyConnectedLayer(320, 50, RELU);
    m_layers[6] = new FullyConnectedLayer(50, 10);
    m_layers[7] = new SoftmaxLayer(10);

    return Network::connect_layers();
}
```

Notice how easily we set an activation function in each layer, and if no activation function is set, the output of the layer will be directly fed into the next layer without passing any activations.

Next, We explain how to train LeNet by leveraging the training data from MNIST data set. The training codes are shown as follows. We carefully add notes on the codes to explain how it works.

```
LeNet lenet;
// Train LeNet in each epoch
for (int i = 0; i < LENET_TOTAL_EPOCH; i++)
{
    // Randomly select training samples for each mini-batch
    MiniBatchSampler batch_sampler;
    batch_sampler.RandomSample(train_dataset.m_images, train_dataset.m_labels, 10, LENET_MINIBATCH_SIZE);
    const std::vector<Tensor> &batch_samples = batch_sampler.GetBatchSampleTensors();
    const std::vector<Tensor> &batch_labels = batch_sampler.GetBatchLabelTensors();

    //  Train LeNet in each mini-batch
    for (int j = 0; j < (int)batch_samples.size(); j++)
    {
        // Forward propagation
        const Tensor &input = batch_samples[j];
        Tensor output;
        lenet.Forward(input, output);

        // Compute loss
        NLLLoss loss;
        double error = loss.Forward(output, batch_labels[j]);

        // Backward Propagation
        Tensor gradient = loss.Backward(error, output, batch_labels[j]);
        lenet.Backward(gradient);

        // Update training coefficients - use Momentum
        SGDOptimizer::MOMENTUM = 0.5f;
        lenet.Update(LENET_LEARNING_RATE, SGD);
    }
}
```

So that's all! Things become super easy when using MoonRiver to create a new network and train it efficiently.

4 Experimental Results

In this section we demonstrate the effectiveness of MoonRiver by training and testing two networks: LeNet and Auto-Encoder. The former is for image classification, whereas the letter is for image generation, or you can view it in another perspective, image dimension reduction.

4.1 LeNet

The goal of LeNet is for each input handwritten image to predict which digit it belongs to. So it can be considered as solving an image classification problem. Here is our setting in training LeNet:

- Training Set: 60,000 MNIST images
- Mini-batch Size: 64
- Total Epochs: 10
- Optimizer: Momentum
- Cost Function: Negative Log Likelihood

We test the trained model with 10,000 testing images from MNIST dataset, which are not included in the training set. The testing/classification results are shown as follows:

- Negative Log Likelihood Loss: 0.038
- Classification Accuracy: 98.97% (9,897/10,000)

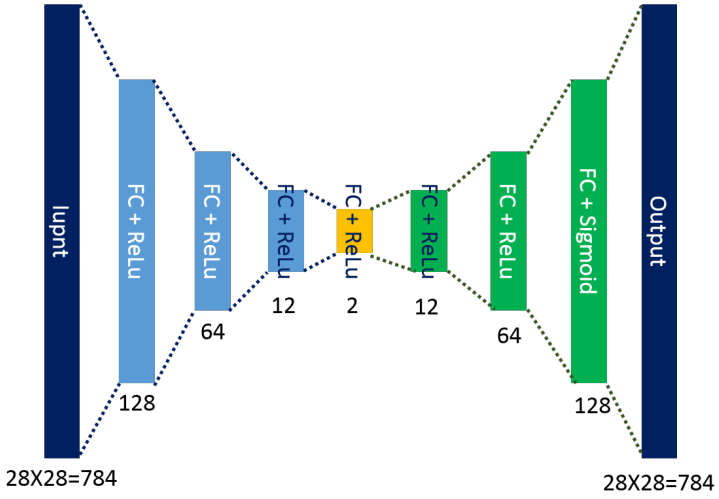

Figure 2: The network architecture of Auto-Encoder.

4.2 Auto-Encoder

Another network we used to test MoonRiver is auto-encoder. The main goal of auto-encoder is to encode an image as a code which is usually has lower dimension than the original image(Notice the term "code" used here means low dimensional representation of a image instead of programming code we used before). The idea is to look for an encoding function to reduce the image dimension from high to low, and another decoding function to reconstruct the high dimensional image from the low dimensional code. The encoding and decoding functions are trained through an end to end neural network training. The auto-encoder architecture we design is shown in Figure.2. Note the yellow layer in the middle of the network is code layer, which only has two dimensions.

Here is the setting of training the auto-encoder:

- Training Set: 60,000 MNIST images
- Mini-batch Size: 64
- Total Epochs: 10
- Optimizer: Adam
- Cost Function:
  - Mean Square Error
  - Mean Square Error + Code L2 norm regularization

Notice that we use two different cost functions to train the auto-encoder. The below is the result of using only mean square error. We show the result by drawing the codes, which are only 2-dimensional, on a 2D image, called code map. It shows significant clustered effect where the codes belongs to the same digits are clustered in the code map.

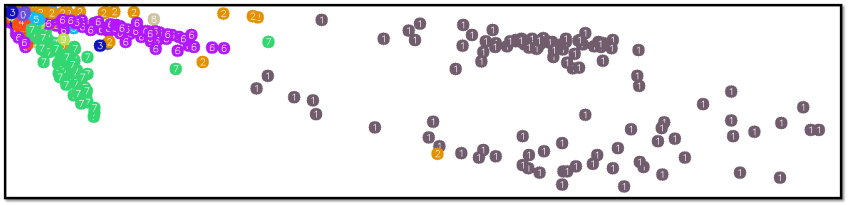

Figure 3: The code map from the training where we only use mean square error as the cost function.

But notice the codes could be highly distributed in the code map if we only use mean square error as the cost function because they can be any values without any penalties. So we try another cost functions which regularizes L2 norm of codes and hopefully could get the map where all codes are centralize around the origin. It works! The result is shown in the following image, which the codes are more evenly distributed in the 2D space.

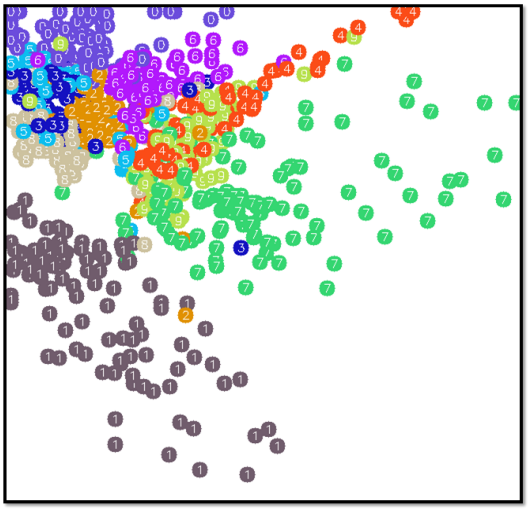

Figure 4: The code map where we use mean square error plus code regularization as the cost function.

In the end, we show some sample reconstruction image results in Figure.5. These images are generated by passing the input images (at top tow) to the trained auto-encoder and then retrieved from the end of the network. The reconstruction images should be as close as possible to the input image (Note the criteria is exactly the cost function we call it mean square error before).

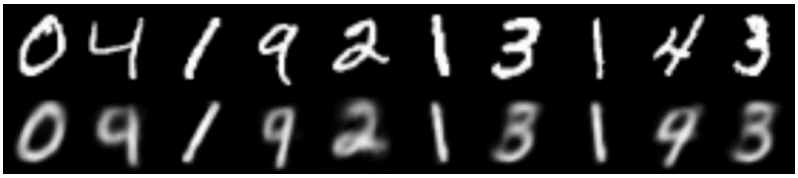

Figure 5: The sample reconstruction results. (Top row) Ground truth. (Bottom Row) Reconstructed images.

5 Conclusion and Future Works

We design a new deep neural network framework, MoonRiver, from scratch. It was implemented in C++. It is lightweight and supports any OSes where have standard C++ compilers. It also let users could easily create a customized network and train it efficiently. There are still lots of extensions we can do in the future. Here we list some of them:

- Support GPU acceleration
- Support Recurrent Neural Network, like LSTM
- Support GAN
- Convert existed trained network, like AlexNet, VGG-Net, or ResNet, into MoonRiver accepted network format

MoonRiver is the start instead of the end. We hopefully in the future we could look for more essential advancement in deep neural network based on the good code base we have already developed.