CS4420 Programming Project 3 (Points: 30)

Weiran Wang
University of Iowa
weiran-wang@uiowa.edu

Abstract

In this project, you will implement the deep feed-forward neural networks, which will be used as the acoustic model for Connectionist Temporal Classification (CTC). This project can be done with less than 100 lines of python code.

1 Task I: Forward and backward functions

Now that we have derived the forward and backpropagation procedures of feedforward neural networks, it is time to implement these algorithms with the `numpy` library.

I have provided a template of the implementation in the file `dnn.py`. You need to implement:

- The `compute_ce_loss()` function, which computes the cross-entropy loss between the network output and ground truth labels, for a minibatch of samples. Additionally, we have an input vector `loss_mask`, which indicates the valid samples (with value 1.0) versus padded samples (with value 0.0) in the minibatch. This function returns the averaged cross-entropy loss over *valid* samples, and its gradient with respect to the network output.
- The class `FeedForwardNetwork`.
  - The constructor `__init__()`: takes as input the input dimension of the data, the output dimension, number of hidden layers, hidden layer width (the same for all hidden layers). You need to initialize `self.weights` and `self.biases`, which are lists storing weights and biases (`numpy` arrays) of each layer. Weights are initialized with uniform random values from the range $[-0.05, 0.05]$, while biases are initialized with 0. The length of each list is `num_hidden_layers + 1`.
  - The `forward()` function: takes as input a minibatch of sample, stored in a `numpy` array of the shape $[d_{in}, batch]$. This function returns an array of the shape $[d_{out}, batch]$ containing the final output (at the last layer), and a list of arrays containing the hidden activations at each hidden layer. **The network uses the ReLU activation at all hidden layers, but no activation at the output layer.**
  - The `backward()` function: takes as input the network output, the hidden activations, the gradient of the loss with respect to the output, and the loss mask. The function returns the list of gradients of loss with respect to the weights of each layer, and the list of gradients of loss with respect to the biases of each layer. Since the loss is averaged cross-entropy loss over valid training samples, the gradient shall be consistent in that regard: invalid or padded samples should be treated as non-existent in the gradient calculation.
  - I have already provided you the functions for updating the network's weights and biases, saving and restoring models, and making predictions for classification. They are used in Task II.

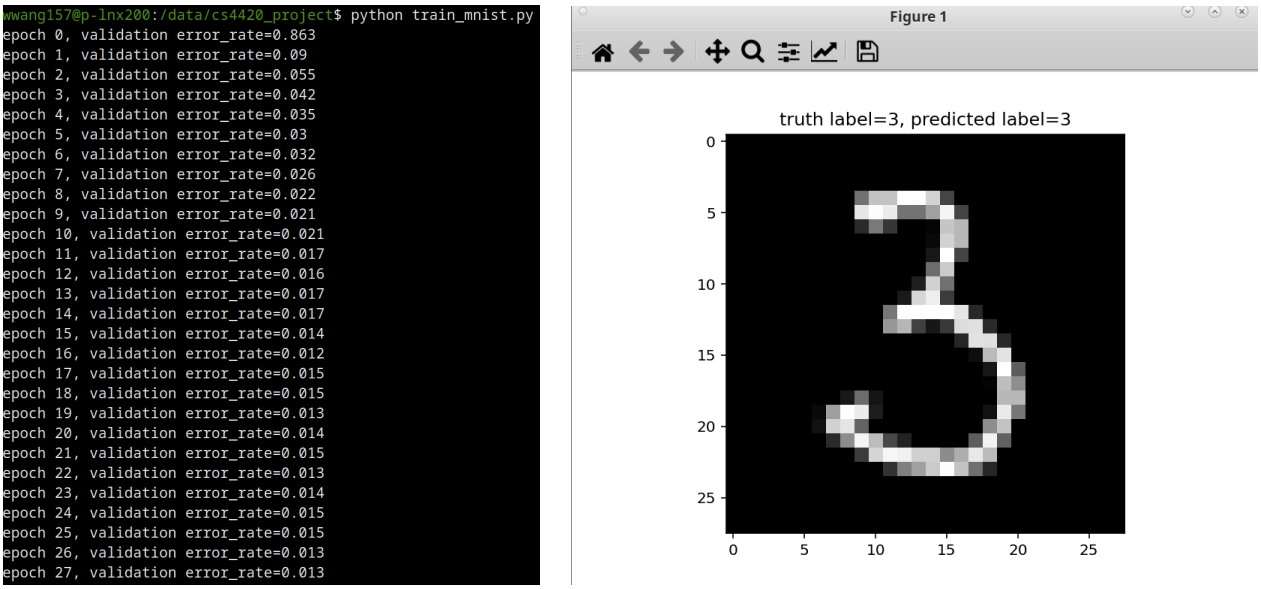

Figure 1: Left: Training log. Right: the neural network at work.

2 Task II: Experimenting with `mnist` classification

The `mnist` dataset is a classical machine learning dataset[1], containing 28x28 greyscale images of handwritten digits 0-9. And we use it to test the neural network we have built.

I have provided the full implementations of the training and test procedures, in `train_mnist.py` and `test_mnist.py` respectively. I have also implemented an input generator in `mnist_input_generator.py`, similar to the one in Project 1, although the `mnist` one is slightly simpler, without the need to handle the sequence structure. We flattern each image as a vector of 784 dimensions, and normalized the pixel values to the range of $[0, 1]$.

Each iteration of the training loop involves:

- Loading a minibatch of training samples, by `train_iter.next()`.
- Forward the neural network with current weights w_t, by `network.forward()`.
- Compute the cross-entropy loss $L(w_t)$ using network output and ground truth labels, and the gradient of loss with respect to network output, with `compute_ce_loss()`.
- Perform back-propagation to compute the gradient of loss with respect to the current network weights, $\nabla L(w_t)$.
- Compute weight update. Here I am using an algorithm called stochastic gradient with momentum (SGD-M): with momentum coefficient $\beta \in (0, 1)$, learning rate $\eta > 0$, $m_0 = 0$, we compute the new model w_{t+1} as

$$m_{t+1} = \beta \cdot m_t + \nabla L(w_t),$$
$$w_{t+1} = w_t - \eta \cdot m_{t+1}.$$

  In other words, in SGD-M, the update is not simply along the direction of stochastic gradient, we also move a bit along the direction of the previous update. In practice, momentum often greatly accelerates learning, if properly tuned.

[1] https://yann.lecun.com/exdb/mnist/.

At the end of each epoch (which means one pass over the training set), we apply the model to validation set to get its validation accuracy; this gives us an idea of the training progress. In fact we could stop training when we observe that the validation performs is converging or deteriorating.

In this task, you need to play around with the hyper-parameters, including batch size, learning rate, number of training epochs, momentum, network architecture (number of hidden layers, and hidden layer width), to find the best model you could on the validation set, and then report the error rate on the test set.

With the default hyper-parameters, I ran `train_mnist.py` and within a matter of minutes, I obtained a model with 1.3% classification error rate on the validation set. The training log, which contains the validation error rate at the end of each epoch is shown in Figure 1 (left). I then ran `test_mnist.py` and the saved model achieves 1.95% error rate on test data, which means the network only makes about 1 error for every 50 images of handwritten digits! You can verify that your model works well by checking some test images and predictions from the model, as shown in Figure 1 (right).

This is the power of machine learning.

3 Submission

You should submit

- For Task I, the file `dnn.py` with your implementations of forward and backward functions. Test your implementation by running `dnn_test.py` and passing the assertions without modifying the file.
- For Task II, the updated `test_mnist.py` (since you may have modified the model architecture), and your saved model `mnist_model.pkl`. We will run your model to get your accuracy.

.