Homework 2: Two-layer perceptron

Hai Dinh (19960331-4494)

1. Method

Based on my experimentation, it's enough to have 8 neurons in the first hidden layer, and 16 neurons in the 2nd hidden layer. The learning rate can be set at 0.02. The weights are initialized with Gaussian distribution with mean 0, and 1 as the standard deviation. The thresholds are initialized with zeros (following the advice from the book). With this setup, the percentron is able to predict with less than 0.12 error rates on the validation approximately after 25 epochs.

The implementation closely follows "Stochastic Gradient Descent" algorithm from Chapter 6 in the course book, with some slight modifications here and there related to the notations and also the dimensionality of different matrix quantities. Note that all computations in the implementation have been vectorized using numpy for faster performance (this goes for both training and validation).

To explain the vectorizations done during the training of the perceptron (note that validation is closely similar), let's introduce some notations. For each layer l after the input layer, let:

- $N^{(l)}$ be the number of neurons/nodes in layer l.
- $W^{(l)}$ be the weight matrix of shape $N^{(l)} \times N^{(l-1)}$.
- $T^{(l)}$ be the threshold vector of shape $N^{(l)} \times 1$.
- $X^{(l)}$ be the input matrix of shape $N^{(l-1)} \times 1$.
- $B^{(l)}$ be the matrix of shape $N^{(l)} \times 1$, storing the local fields of the layer.
- $O^{(l)}$ be the matrix of shape $N^{(l)} \times 1$, storing the outputs of the layer.
- $E^{(l)}$ be the matrix of shape $N^{(l)} \times 1$, storing the errors of the layer.
- $\mathbf{t}$ be the target output (from the dataset) of shape 1×1.
- $\odot$ will be used to indicate Hadamard product (i.e., element-wise multiplication).

Step 1: For layer $l = 1, \ldots, L$, forward propagate by:

- $B^{(l)} := W^{(l)} X^{(l)} - T^{(l)}$
- $O^{(l)} := \tanh\left(B^{(l)}\right)$
- $X^{(l+1)} := O^{(l)}$

Step 2: Compute the error of the output layer (L) by:

- $E^{(L)} := \left(\mathbf{t} - O^{(L)}\right) \odot \left(1 - \tanh^2\left(B^{(L)}\right)\right)$

Step 3: For layer $l = L, \ldots, 2$, backward propagate the errors by:

- $E^{(l-1)} := \left(\left(W^{(l)}\right)^\intercal E^{(l)}\right) \odot \left(1 - \tanh^2\left(B^{(l-1)}\right)\right)$

Step 4: For layer $l = 1, \ldots, L$, update weights and thresholds using the computed errors:

- $W^{(l)} := W^{(l)} + \eta E^{(l)} \left(X^{(l)}\right)^\intercal$
- $T^{(l)} := T^{(l)} - \eta E^{(l)}$

2. Code

```
import argparse
import numpy as np
```

```
import matplotlib.pyplot as plt

# Load training data
train_set = np.genfromtxt("training_set.csv", delimiter=",").T
n_train = train_set.shape[1]

# Load validation data
val_set = np.genfromtxt("validation_set.csv", delimiter=",").T
n_val = val_set.shape[1]
X_val = val_set[:-1, :]
Y_val = val_set[-1:, :]
input_dim = X_val.shape[0]

# Pre-defined constants
accepted_error_rate = 0.12
eta = 0.02
n_epochs = 1000
n_layers = 3
n_dims = [input_dim, 8, 16, 1] # number of dimensions in all layers (including input layer)

# Initialize weights and thresholds for all layers
W = [np.random.normal(0, 1, size=(n_dims[l], n_dims[l-1])) for l in range(1, n_layers+1)]
T = [np.zeros(shape=(n_dims[l], 1)) for l in range (1, n_layers+1)]

def tanh_derivative(values):
    return (1 - np.tanh(values)**2)

def run_perceptron(inputs, targets, is_training=False):
    X = [inputs] # store inputs for all layers
    B = []       # store local fields for all layers
    O = []       # store outputs for all layers
    E = []       # store errors for all layers

    # Forward propagate
    for l in range(n_layers):
        B.append(W[l] @ X[-1] - T[l])
        O.append(np.tanh(B[-1]))

        if l != n_layers - 1:
            X.append(O[-1])

    # Backward propagate only during training (assuming batch size of 1)
    if is_training and inputs.shape[1] == 1:
        E.insert(0, ((targets - O[-1]) * tanh_derivative(B[-1])))

        for l in range(-1, -n_layers, -1):
            E.insert(0, (W[l].T @ E[0] * tanh_derivative(B[l-1])))

        for l in range(n_layers):
            W[l] += eta * E[l] @ X[l].T
            T[l] -= eta * E[l]

    return O[-1]
```

```
def main(args):
    np.random.seed(args.seed)

    for epoch in range(1, n_epochs + 1):
        np.random.shuffle(train_set.T)
        X_train = train_set[:-1, :]
        Y_train = train_set[-1:, :]

        # Training
        for mu in range(n_train):
            run_perceptron(X_train[:,mu,None], Y_train[:,mu,None], is_training=True)

        # Validating
        Y_pred = np.sign(run_perceptron(X_val, Y_val, is_training=False))
        error_rate = (Y_pred != Y_val).sum() / n_val

        # Print summary and check stopping criteria
        print("[Epoch " + str(epoch) + "]", "Error rate:", error_rate)
        if error_rate <= accepted_error_rate: break

    for l in range(n_layers):
        np.savetxt("{}/w{}.csv".format(args.outdir, l+1), W[l], delimiter=",")
        np.savetxt("{}/t{}.csv".format(args.outdir, l+1), T[l], delimiter=",")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Experiment with 2-layer perceptron")
    parser.add_argument("--outdir", "-o", type=str, default=".", help="Out directory")
    parser.add_argument("--seed", "-s", type=int, default=None, help="Seed for random generator")
    main(parser.parse_args())
```

3. Results

```
[Epoch 1] Error rate: 0.1442
[Epoch 2] Error rate: 0.1364
[Epoch 3] Error rate: 0.1378
[Epoch 4] Error rate: 0.135
[Epoch 5] Error rate: 0.1338
[Epoch 6] Error rate: 0.1294
[Epoch 7] Error rate: 0.1302
[Epoch 8] Error rate: 0.129
[Epoch 9] Error rate: 0.1292
[Epoch 10] Error rate: 0.1398
[Epoch 11] Error rate: 0.1262
[Epoch 12] Error rate: 0.1284
[Epoch 13] Error rate: 0.1322
[Epoch 14] Error rate: 0.1338
[Epoch 15] Error rate: 0.1264
[Epoch 16] Error rate: 0.1326
[Epoch 17] Error rate: 0.1296
[Epoch 18] Error rate: 0.1288
[Epoch 19] Error rate: 0.128
[Epoch 20] Error rate: 0.1258
[Epoch 21] Error rate: 0.1318
```

```
[Epoch 22] Error rate: 0.1222
[Epoch 23] Error rate: 0.1222
[Epoch 24] Error rate: 0.1206
[Epoch 25] Error rate: 0.1188
```