Homework 1

Student Number: 118033910019
Name: Xingyi Wang

Problem 1. One variation of the perceptron rule is:

$$W^{new} = W^{old} + \alpha e p^T$$

$$b^{new} = b^{old} + \alpha e$$

where α is the learning rate. Prove convergence of this algorithm. Does the proof require a limit on the learning rate? Please explain.

Solution. Take a Single-Neuron Perceptron $a = hardlim({}_1\mathbf{w}^T\mathbf{p} + b)$ as an example. Let

$$\mathbf{x} = \begin{bmatrix} {}_1\mathbf{w} \\ b \end{bmatrix}, \tag{1}$$

and

$$\mathbf{z}_q = \begin{bmatrix} \mathbf{p}_q \\ 1 \end{bmatrix}. \tag{2}$$

Then we have

$$n =_1 \mathbf{w}^T\mathbf{p} + b = \mathbf{x}^T\mathbf{z}. \tag{3}$$

Now, the perceptron learning rule can be expressed as

$$\mathbf{x}^{new} = \mathbf{x}^{old} + \alpha e \mathbf{z}. \tag{4}$$

If $e = 0$, $\mathbf{x}$ will not be updated. Otherwise, for iteration k,

$$\mathbf{x}(k) = \mathbf{x}(k-1) + \alpha \mathbf{z}'(k-1), \tag{5}$$

where $\mathbf{z}'(k-1) \in \{\mathbf{z}_1, \mathbf{z}_2, ..., \mathbf{z}_Q, -\mathbf{z}_1, -\mathbf{z}_2, ..., -\mathbf{z}_Q\}$. Assume that there exists a weight vector $\mathbf{x}^*$ that can correctly categorize all Q input vectors. Then

$$\mathbf{x}^{*T}\mathbf{z}_q > \delta > 0 \text{ if } t_q = 1, \tag{6}$$

and

$$\mathbf{x}^{*T}\mathbf{z}_q < -\delta < 0 \text{ if } t_q = 0. \tag{7}$$

To prove the convergence of this algorithm, we are going to find the upper and lower bounds on the length of the weight vector at each stage of the algorithm.
Without loss of generality, assume that the algorithm is initialized with the zero weight vector: $\mathbf{x}(0) = \mathbf{0}$. Then, after k iterations, we have

$$\mathbf{x}(k) = \alpha(\mathbf{z}'(0) + \mathbf{z}'(1) + ... + \mathbf{z}'(k-1)). \tag{8}$$

Take the inner product of the solution weight vector with the weight vector at iteration k, we have

$$\mathbf{x}^{*T}\mathbf{x}(k) = \alpha(\mathbf{x}^{*T}\mathbf{z}'(0) + \mathbf{x}^{*T}\mathbf{z}'(1) + ... + \mathbf{x}^{*T}\mathbf{z}'(k-1)). \tag{9}$$

From Eq. 6 and Eq. 7, we have

$$\mathbf{x}^{*T}\mathbf{z}'(i) > \delta. \tag{10}$$

Therefore

$$\mathbf{x}^{*T}\mathbf{x}(k) > \alpha k\delta. \tag{11}$$

From the Cauchy-Schwartz inequality,

$$(\mathbf{x}^{*T}\mathbf{x}(k))^2 \leq \|\mathbf{x}^*\|^2\|\mathbf{x}(k)\|^2, \tag{12}$$

where $\|\mathbf{x}\|^2 = \mathbf{x}^T\mathbf{x}$.
Combine Eq. 11 and Eq. 12, then we can put a lower bound on the squared length of the weight vector at iteration k:

$$\|\mathbf{x}(k)\|^2 \geq \frac{(\mathbf{x}^{*T}\mathbf{x}(k))^2}{\|\mathbf{x}^*\|^2} > \frac{(\alpha k\delta)^2}{\|\mathbf{x}^*\|^2}. \tag{13}$$

Next, we want to find an upper bound for the length of the weight vector. At iteration k, the weight vector is changed:

$$\begin{aligned}\|\mathbf{x}(k)\|^2 &= \mathbf{x}^T(k)\mathbf{x}(k) \\ &= [\mathbf{x}(k-1) + \alpha\mathbf{z}'(k-1)]^T[\mathbf{x}(k-1) + \alpha\mathbf{z}'(k-1)] \\ &= \mathbf{x}^T(k-1)\mathbf{x}(k-1) + 2\alpha\mathbf{x}^T(k-1)\mathbf{z}'(k-1) + \alpha^2\mathbf{z}'^T(k-1)\mathbf{z}'(k-1).\end{aligned} \tag{14}$$

Note that, since the weights would not be updated unless the previous input vector had been misclassified, so

$$\mathbf{x}^T(k-1)\mathbf{z}'(k-1) \leq 0. \tag{15}$$

Then we have

$$\|\mathbf{x}(k)\|^2 \leq \|\mathbf{x}(k-1)\|^2 + \alpha^2\|\mathbf{z}'(k-1)\|^2. \tag{16}$$

We can repeat this process for $\|\mathbf{x}(k-1)\|^2$, $\|\mathbf{x}(k-2)\|^2$, ..., to get

$$\|\mathbf{x}(k)\|^2 \leq \alpha^2(\|\mathbf{z}'(0)\|^2 + ... + \|\mathbf{z}'(k-1)\|^2). \tag{17}$$

Let $\Pi = max\{\|\mathbf{z}'(i)\|^2\}$, then this upper bound can be simplified to

$$\|\mathbf{x}(k)\|^2 \leq \alpha^2 k\Pi. \tag{18}$$

Now we have an upper bound (Eq. 18) and a lower bound (Eq. 13) on the squared legnth of the weight vector at iteration k. Combine them and we have

$$\alpha^2 k\Pi \geq \|\mathbf{x}(k)\|^2 > \frac{(\alpha k\delta)^2}{\|\mathbf{x}^*\|^2} \text{ or } k < \frac{\|\mathbf{x}^*\|^2\Pi}{\delta^2} \tag{19}$$

Because k has an upper bound, this means that the weights will only be changed a finite number of times. Therefore, the perceptron learning rule will converge in a finite number of iterations. And since there is no α in the upper bound of k, this proof requires no limit on the learning rate.

Problem 2. Suppose the output of each neuron in a multipayer perceptron network is

$$x_{kj} = f(\sum_{i=1}^{N_{k-1}} (u_{kji} x_{k-1,i}^2 + v_{kji} x_{k-1,i}) + b_{kj})$$

for $k = 2, 3, ..., M$ and $j = 1, 2, ..., N_k$, where both u_{kji} and v_{kji} are the weights connecting the i-th unit in the layer $k-1$ to the j-th unit in the layer k, b_{kj} is the bias of the j-th unit in the layer k, N_k is the number of units if the k $(1 \leq k \leq M)$, and $f(\cdot)$ is the sigmoidal activation function.
The structure of the unit is shown as the following figure, and this network is called multi-layer quadratic perceptron (MLQP).

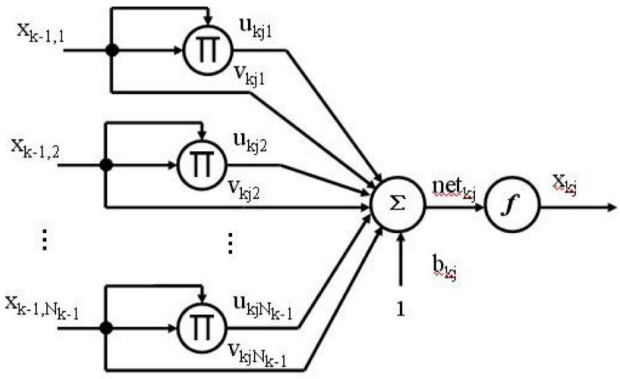

Figure 1: The structure of MLQP

Figure 1: The structure of MLQP

Please derive the back-propagation algorithms for MLQPs in both on-line learning and batch learning ways.

Solution.

1. In the on-line learning, weights are updated after presenting each training example. Assume an input vector is passed forward to the output layer, and we get the total instantaneous error energy $\mathcal{E}$, where the cost function is

$$\mathcal{E} = \frac{1}{2} \sum_{j=1}^{N_M} e_{Mj}^2 \tag{20}$$

. We want to use back-propagation algorithms to compute Δu_{kji}, Δv_{kji}, and Δb_{kj}. Take Δu_{kji} as an example, according to the *chainrule*, we have:

$$\Delta u_{kji} = -\eta \frac{\partial \mathcal{E}}{\partial u_{kji}} = -\eta \frac{\partial \mathcal{E}}{\partial x_{kj}} \frac{\partial x_{kj}}{\partial u_{kji}}. \tag{21}$$

Let

$$t_{kj} = \sum_{i=1}^{n_{k-1}} (u_{kji} x_{k-1,i}^2 + v_{kji} x_{k-1,i}) + b_{kj}, \tag{22}$$

then $x_{kji} = f(t_{kji})$. So Δu_{kji} can be updated as

$$\Delta u_{kji} = -\eta \frac{\partial \mathcal{E}}{\partial x_{kj}} \frac{\partial x_{kj}}{\partial t_{kj}} \frac{\partial t_{kj}}{\partial u_{kji}}. \tag{23}$$

Define *local gradient* δ_{kj} as

$$\delta_{kj} = -\frac{\partial \mathcal{E}}{\partial t_{kj}}, \tag{24}$$

then Δu_{kji} can be updated as

$$\Delta u_{kji} = -\eta \frac{\partial \mathcal{E}}{\partial t_{kj}} \frac{\partial t_{kj}}{\partial u_{kji}} = \eta \delta_{kj} x_{k-1,i}^2. \tag{25}$$

Similarly, we can have:

$$\Delta v_{kji} = -\eta \frac{\partial \mathcal{E}}{\partial t_{kj}} \frac{\partial t_{kj}}{\partial v_{kji}} = \eta \delta_{kj} x_{k-1,i}, \tag{26}$$

$$\Delta b_{kj} = -\eta \frac{\partial \mathcal{E}}{\partial t_{kj}} \frac{\partial t_{kj}}{\partial b_{kj}} = \eta \delta_{kj}. \tag{27}$$

Now, let us have a discussion about δ_{kj}.

(a) If layer k is the output layer ($k = M$), then we have

$$\begin{aligned} \delta_{Mj} &= -\frac{\partial \mathcal{E}}{\partial t_{Mj}} \\ &= -\frac{\partial \mathcal{E}}{\partial e_{Mj}} \frac{\partial e_{Mj}}{\partial x_{Mj}} \frac{\partial x_{Mj}}{\partial t_{Mj}} \\ &= e_{Mj} f'(t_{Mj}) \\ &= e_{Mj} f(t_{Mj})(1 - f(t_{Mj})) \end{aligned} \tag{28}$$

where $f' = f(1 - f)$ since $f(\cdot)$ is the sigmoidal activation function.

(b) If layer k is not the output layer ($k < M$), then we have

$$\begin{aligned}
\delta_{kj} &= -\frac{\partial \mathcal{E}}{\partial t_{kj}} \\
&= -\sum_{l=1}^{N_{k+1}} \frac{\partial \mathcal{E}}{\partial t_{k+1,l}} \frac{\partial t_{k+1,l}}{\partial x_{kj}} \frac{\partial x_{kj}}{\partial t_{kj}} \\
&= -\sum_{l=1}^{N_{k+1}} \delta_{k+1,l}(2u_{k+1,lj}x_{kj} + v_{k+1,lj}) f'(t_{kj}) \\
&= -\sum_{l=1}^{N_{k+1}} \delta_{k+1,l}(2u_{k+1,lj}x_{kj} + v_{k+1,lj}) f(t_{Mj})(1 - f(t_{Mj}))
\end{aligned} \tag{29}$$

So δ_{kj} can be calculated by the iteration between layers, and the iteration returns when it reaches the output layer.

2. In the batch learning, weights are updated after each epoch only. So the cost function is

$$\mathcal{E} = \frac{1}{2A} \sum_{a=1}^{A} \sum_{j=1}^{N_M} e_{Mj}^2 \tag{30}$$

where A is the number of training samples. Then the only change is the local gradient at the output layer. Now we have

$$\begin{aligned}
\delta_{Mj} &= -\frac{\partial \mathcal{E}}{\partial t_{Mj}} \\
&= -\frac{\partial \mathcal{E}}{\partial e_{Mj}} \frac{\partial e_{Mj}}{\partial x_{Mj}} \frac{\partial x_{Mj}}{\partial t_{Mj}} \\
&= \frac{\eta}{A} \sum_{a=1}^{A} e_{Mj} f'(t_{Mj}) \\
&= \frac{\eta}{A} \sum_{a=1}^{A} e_{Mj} f(t_{Mj})(1 - f(t_{Mj}))
\end{aligned} \tag{31}$$

Problem 3. In problem 3, you should program a deep neural network with TensorFlow or PyTorch to solve an emotion recognition task.
The deep neural network for this task is a three layer network: an input layer, a hidden layer, and an output layer. The neurons in the input layer should be the same as input feature dimensions. The number of hidden neurons is a hyperparameter which is chosen by yourself. And there are 3 output units for 3 emotions.
The provided data contains differential entropy (DE) features extracted from emotional EEG signals. Emotions include positive, neutral, and negative.
Run your code to classify three emotion states (positive, neutral, negative) and compare the training time and generalization performance of different hidden units and learning rates.

Solution. In this task, there are 499 training samples and () testing samples. The input feature is 310-dimension.
So I designed a three-layer MLP model: the input layer has 310 neurons, which is the same as the input feature dimensions; the number of hidden units is chosen to be some certain numbers; and the output layer has 3 neurons.
The input features are normalized using 'L2-norm'. The labels are one-hot encoded (-1: [1, 0, 0], 0: [0, 1, 0], 1: [0, 0, 1]).
I compared the experimental results under different number of hidden units and different learning rates. The model is trained for 1000 epoches. The results are shown below:

```
# hidden units: 15       learning rate: 0.1     training loss: 0.6652   training accuracy: 0.6834   testing accuracy: 0.6618
# hidden units: 15       learning rate: 0.01    training loss: 0.6164   training accuracy: 1.0000   testing accuracy: 0.6822
# hidden units: 15       learning rate: 0.001   training loss: 0.9796   training accuracy: 0.8818   testing accuracy: 0.8309
# hidden units: 50       learning rate: 0.1     training loss: 1.0013   training accuracy: 0.3387   testing accuracy: 0.3178
# hidden units: 50       learning rate: 0.01    training loss: 0.6257   training accuracy: 1.0000   testing accuracy: 0.6822
# hidden units: 50       learning rate: 0.001   training loss: 0.9013   training accuracy: 0.7916   testing accuracy: 0.6822
# hidden units: 100      learning rate: 0.1     training loss: 1.1786   training accuracy: 0.3387   testing accuracy: 0.3178
# hidden units: 100      learning rate: 0.01    training loss: 1.0986   training accuracy: 0.3166   testing accuracy: 0.3382
# hidden units: 100      learning rate: 0.001   training loss: 0.7319   training accuracy: 1.0000   testing accuracy: 0.6822
# hidden units: 150      learning rate: 0.1     training loss: 1.0986   training accuracy: 0.3387   testing accuracy: 0.3178
# hidden units: 150      learning rate: 0.01    training loss: 1.0986   training accuracy: 0.3387   testing accuracy: 0.3178
# hidden units: 150      learning rate: 0.001   training loss: 0.7304   training accuracy: 1.0000   testing accuracy: 0.6822
# hidden units: 200      learning rate: 0.1     training loss: 1.0986   training accuracy: 0.3387   testing accuracy: 0.3178
# hidden units: 200      learning rate: 0.01    training loss: 1.0986   training accuracy: 0.3387   testing accuracy: 0.3178
# hidden units: 200      learning rate: 0.001   training loss: 0.7823   training accuracy: 0.8818   testing accuracy: 0.6822
```

Figure 2: Experimental results

We can see that, when the number of hidden units is 15 and the learning rate is 0.001, we get the highest testing accuracy: 0.8309. In general, a lower learning rate can guarantee the convergence of the model training, resulting in a higher accuracy. Since we don't have many training samples, a large number of hidden units cannot be well-trained (15 hidden states are enough).