HW5. Neural Networks

Haanvid Lee (hvlee@ai.kaist.ac.kr)

December 4, 2017

1 Introduction

In this assignment, you will design and build the neural network to classify classes in two image datasets. One dataset has images of handwritten digits, and the other one has images of faces.

2 Project Instruction

2.1 NumPy and Anaconda

In this assignment, will use **Python 2.7** and **NumPy**, but not SciPy.

Packages and versions required for this homework are:

- Python 2.7.14: https://www.python.org/
- NumPy 1.13.3: http://www.numpy.org/
- (Recommended) Anaconda: https://www.continuum.io/downloads/

2.2 Neural Network

In this homework, we will implement multilayer perceptron (MLP). Training procedure of the neural network can be divided into two parts: forward propagation and back propagation. During the forward pass, neural activation values from input to output are computed. And at the end of the network, error is obtained by comparing the model output with the true output. By making the backward pass, gradients of the error with respect to (w.r.t.) parameters are calculated and used for learning parameters that minimize the error.

2.2.1 Forward Propagataion

There are three types of layers in a MLP: input layer, hidden layer, and output layer. There could be one or more hidden layers in a MLP. Each layer except the input layer contains several artificial neurons (illustrated as circles in Fig. 1). In a MLP, neural activations in a layer are computed based on the real values assigned to units of the previous layer and the parameters (the arrows in Fig. 1) that connect previous and current layers.

At the beginning of the forward pass, the values of the input data is assigned to the nodes of the input layer (i.e., if you have 3 values in your input data, the values are assigned to the 3 nodes in the input layer that are not neurons). Then, to compute the neural activations of the hidden layer 1, parameters are multiplied to the input layer node values. For each neuron in the hidden layer 1, the received values are summed up to compute the internal neural state as in Eq. (1) (illustrated as the input function of a neuron in Fig. 2).

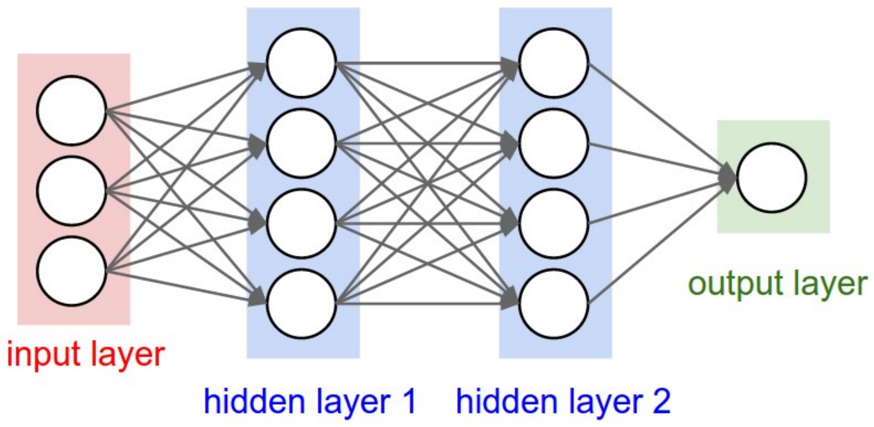

Figure 1: Neural network structure

$$z_{lj} = \mathbf{w}_{lj}^{\top}\mathbf{x}_l + b_{lj}. \tag{1}$$

In the equation, internal neural state of a jth neuron at lth layer (in this case, $l = 1$ since hidden layer 1 is the first layer that is composed of neurons) is represented as z_{lj}. The internal state is computed by multiplying the values in the nodes of its previous layer ($\mathbf{x}_l$ is the vector representing the nodal values) with the parameters that connect the nodes in the previous layer with the jth neuron at lth layer. The vector of parameters are represented as $\mathbf{w}_{lj}$. Additionally, b_{lj} represents the bias for the neuron.

After calculating the internal neural states, the computed values go through activation functions to make the neural activations as in Eq. (2) (see the illustration of the activation function of a neuron in Fig. 2).

$$y_{lj} = a(z_{lj}). \tag{2}$$

Here, y_{lj} denotes the activation value of jth neuron at lth layer, a is the activation function (details on activation functions will be discussed in Section 2.2.2). Same computing processes are repeated to calculate the internal neural states and activations of the following layers. The neural activations of the last layer is the output of the network. Since we were computing activation values in the first hidden layer, $\mathbf{x}_l$ was simply the input values of the network. But for the computation of the following layers, $\mathbf{x}_l$ is the neural activations of the previous layer.

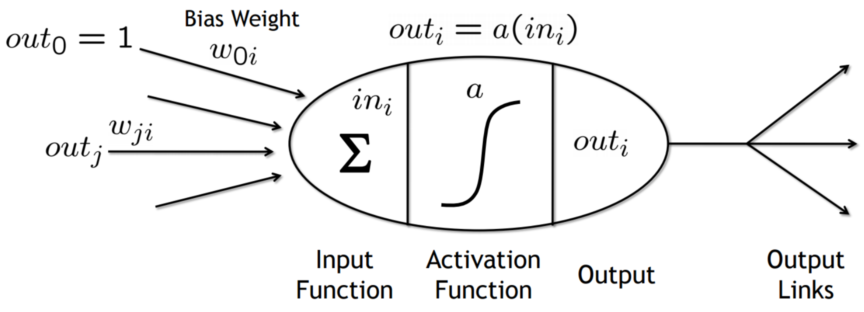

Figure 2: One node of an hidden layer

Since NumPy can be used to do efficient matrix computations, we can exploit this characteristic by computing internal neural states of all neurons at once as in Eq. (3).

$$\mathbf{z}_l = \mathbf{W}_l^\top \mathbf{x}_l + \mathbf{b}_l. \tag{3}$$

Here, $\mathbf{z}_l$ represents the vector that is composed of internal neural states of lth layer. The biases for the neurons are also represented as a vector form ($\mathbf{b}_l$). The weight vector $\mathbf{w}_{lj}$ in Eq. (1) can be stacked by the node index j and form the matrix $\mathbf{W}_l$. You can further take advantage of NumPy by computing the internal neural states of lth layer for multiple input data points ($\mathbf{Z}_l$) as in Eq. (4).

$$\mathbf{Z}_l = \mathbf{X}_l^\top \mathbf{W}_l + \mathbf{B}_l. \tag{4}$$

2.2.2 Activation Functions

Activation functions are usually non-linear functions that give non-linear transformation of internal neural states in a neural network. In this homework, we use three types of activation functions: ReLU, softmax, and sigmoid.

Rectified Linear Unit (ReLU)

ReLU activation is the most popular activation function currently used. The ReLU can be represented as in Eq. (5):

$$a(z) = \max(0, z) \tag{5}$$

Here the function output (neural activation) is 0 if the input (internal neural state) is negative or 0. And the input is the output when the input is positive. ReLU activation function is known to accelerate the convergence during the training phase of a network. This is because it has constant slopes whereas some other activation functions (e.g., sigmoid) have decreasing slopes as the input value for the function gets large. Also, it is very easy to compute. In this homework, we will use this activation function for all neurons in the network except the ones at the output layer.

Softmax

Softmax is mostly used for making categorical outputs at the end of neural networks for multi-class classification tasks. In a classification task, each softmax neuron at the output layer of a network represents a class. And each activation value represent the probability that a input data point belonging to the class represented by the neuron. The softmax function is shown in Eq. (6).

$$a(\mathbf{z}_j) = \frac{\exp(\mathbf{z}_j)}{\sum_i \exp(\mathbf{z}_i)}. \tag{6}$$

In Eq. (6), we can see that the softmax activations in the layer is summed up to 1. Also note that a softmax function is not an element-wise operator: activation at the jth neuron ($\mathbf{y}_j$) is dependent on all internal neural states.

Sigmoid

Sigmoid is usually used as the activation function at the output layer when dealing with a binary classification problem. The form of a sigmoid function is:

$$a(z) = \frac{1}{1 + \exp(-z)} \tag{7}$$

Sigmoid function output ranges from 0 to 1. And only one neuron is used at the output layer when dealing with a binary classification task. The function output represents the probability that the network input data is class 1. To get the probability of the other class, we subtract the activation value from 1: $1 - y_L = 1 - \Pr(\text{label} = 1|x) = \Pr(\text{label} = 2|x)$.

Linear

Linear function is the most simplest form of a activation function as shown in Eq. (8).

$$a(z) = z. \tag{8}$$

Using this function is same with not using any activation functions. This function is normally used for the output layer when the neural network is used for a regression task that needs predicting real values.

2.2.3 Back Propagation

After making network outputs, the outputs are compared with the true outputs by calculating errors using their values. Then, the errors are used to update the weights and biases of the neural network.

Different error functions (loss functions) are used for binary and multi-class classification tasks. For a binary classification, we use binary cross-entropy loss:

$$\text{Loss}(y, y_L) = -\left[y \log(y_L) + (1 - y) \log(1 - y_L)\right], \tag{9}$$

where y represents the true output and is 0 for "class 1" and 1 for "class 2". Also, y_L is the output of the nerual network with L layers that are composed of neurons.

For a multi-class classification, we use categorical cross-entropy loss:

$$\text{Loss}(\mathbf{y}, \mathbf{y}_L) = -\left[\sum_j y_j \log(y_{Lj})\right], \tag{10}$$

where, $\mathbf{y}$ is a vector which elements have 0 value except the true label, which has 1.

With the errors calculated from the loss functions, weights and biases of a neural network are updated in a way that minimizes the errors. To find the direction of the update, gradients of the error w.r.t. the weights and biases are calculated.

Let's first think about the error gradient w.r.t. the weight in the last layer L. If the activation function is element-wise (so if it is not a softmax,), by chain rule, you can write:

$$\frac{\partial \text{Loss}}{\partial \mathbf{W}_L} = \sum_j \frac{\partial \text{Loss}}{\partial y_{Lj}} \frac{\partial y_{Lj}}{\partial z_{Lj}} \frac{\partial z_{Lj}}{\partial \mathbf{W}_L}$$

The beginning of the result of the chain rule is to calculate the gradient of $\mathbf{y}_L$ to the loss.

For binary cross-entropy loss:

$$\frac{d\text{Loss}}{dy_L} = -\left[\frac{y}{y_L} - \frac{1-y}{1-y_L}\right]$$

The second term is the derivation of each activation function. Graidents of linear and ReLU activation function is omitted. Gradient of the sigmoid function is:

$$\frac{\partial y_L}{\partial z_L} = \frac{1}{1 + \exp(-z_L)}\left(1 - \frac{1}{1 + \exp(-z_L)}\right) = \text{Sigmoid}(z_L)(1 - \text{Sigmoid}(z_L))$$

Also you can use the fact $y_L = \text{Sigmoid}(z_L)$,

$$\frac{\partial y_L}{\partial z_L} = y_L(1 - y_L)$$

For the softmax function, the chain rule becomes:

$$\frac{\partial \text{Loss}}{\partial \mathbf{W}_L} = \sum_k \frac{\partial \text{Loss}}{\partial y_{Lk}} \sum_j \frac{\partial y_{Lk}}{\partial z_{Lj}} \frac{\partial z_{Lj}}{\partial \mathbf{W}_L}$$

For categorical cross-entropy loss, error gradient w.r.t. $\mathbf{y}_{Lk}$ is:

$$\frac{\partial \text{Loss}}{\partial y_{Lk}} = -\frac{y_k}{y_{Lk}}$$

Gradient of the softmax function is:

$$\frac{\partial y_{Lk}}{\partial z_{Lj}} = \begin{cases} y_{Lj}(1 - y_{Lj}), (\text{ if } j = k) \\ -y_{Lj} y_{Lk}, (\text{ if } j \neq k) \end{cases} \tag{11}$$

For both cases of binary and categorical cross entropy losses, you can easily calculate the third term of the chain rule, $\frac{\partial z_{Lj}}{\partial \mathbf{W}_L}$ from Eq. (1)

$$\frac{\partial z_{Lj}}{\partial \mathbf{w}_{Lj}} = \mathbf{x}_L \tag{12}$$

You can get the gradient of bias in a similar way (actually only the third term is changed). And for the gradients in the previous layers, you can repeat this process from the *(L-1)*th layer to the 1st layer.

2.2.4 Back Propagation - Easier Way

Lets' change the chain rule a little.

$$\frac{\partial \text{Loss}}{\partial \mathbf{W}_l} = \sum_j \frac{\partial \text{Loss}}{\partial z_{lj}} \frac{\partial z_{lj}}{\partial \mathbf{W}_l}$$

Then, define the first term of this changed chain rule as δ:

$$\delta_{lj} = \frac{\partial \text{Loss}}{\partial z_{lj}}$$

It is easy to know that:

$$\frac{\partial \text{Loss}}{\partial \mathbf{W}_l} = \mathbf{x}_l \delta_l^\top \tag{13}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{b}_l} = \delta_l \tag{14}$$

You may want to make it as a matrix form:

$$\frac{\partial \text{Loss}}{\partial \mathbf{W}_l} = \mathbf{X}_l^\top \delta_l$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{b}_l} = \sum_j \delta_{lj}$$

How about propagating back? For δ_{l-1},

$$\delta_{(l-1)k} = \frac{\partial \text{Loss}}{\partial z_{(l-1)k}} = \sum_j \frac{\partial \text{Loss}}{\partial z_{lj}} \frac{\partial z_{lj}}{\partial z_{(l-1)k}} = \sum_j \delta_{lj} \frac{\partial z_{lj}}{\partial z_{(l-1)k}}$$

$$\begin{aligned} \frac{\partial z_{lj}}{\partial z_{(l-1)k}} &= \frac{\partial [\mathbf{w}_{lj}^\top \mathbf{x}_l + b_{lj}]}{\partial z_{(l-1)k}} \\ &= \frac{\partial [\mathbf{w}_{lj}^\top a(\mathbf{z}_{l-1}) + b_{lj}]}{\partial z_{(l-1)k}} \\ &= w_{ljk} * a'(z_{(l-1)k}) \end{aligned}$$

Therefore,

$$\delta_{(l-1)k} = a'(z_{(l-1)k}) \sum_j \delta_{lj} w_{ljk} \tag{15}$$

$$\delta_{(l-1)} = \mathbf{W}_l \delta_l \circ a'(\mathbf{z}_{(l-1)}) \tag{16}$$

Be aware that in this expression, δ_l is a column vector which contains the error of layer l for current input data $\mathbf{x}_l$. And, $\circ$ denotes element-wise multiplication.

If we have several data at once so the input data $\mathbf{X}_l$ is now a matrix, then δ_l also should be a matrix which contains the error of layer l for each data. This is a matrix form of *Eq.* (16) with several input data:

$$\delta_{(l-1)} = \delta_l \mathbf{W}_l^\top \circ a'(\mathbf{z}_{(l-1)})$$

For ReLU, a' can be defined as:

$$\begin{aligned} \mathrm{ReLU}'(z_{(l-1)k}) &= 1 && (\text{If } y_{(l-1)k} = x_{lk} > 0) \\ &= 0 && (\text{If } y_{(l-1)k} = x_{lk} = 0) \end{aligned}$$

So, if you save all value of nodes $\mathbf{x}_l$, and can calculate δ_L, weights and biases can be updated like Eq. (13) and Eq. (14), and delta can be updated like Eq. (16)

How about δ_L? For the cases (binary cross-entropy + sigmoid) and (categorical cross-entropy + softmax), by calculating $\sum_k \frac{\partial \mathrm{Loss}}{\partial y_{Lk}} \frac{\partial y_{Lk}}{\partial z_{Lj}}$ by hand, you can derive that

$$\delta_L = \mathbf{y}_L - \mathbf{y}$$

After calculating all the gradients of weights and biases, you can use gradient descent to update the parameters to make the loss lower:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \frac{\partial \mathrm{Loss}}{\partial \mathbf{W}}$$

$$\mathbf{b} \leftarrow \mathbf{b} - \alpha \frac{\partial \mathrm{Loss}}{\partial \mathbf{b}}$$

2.3 Batch Update

So far we have assumed that there is only one training data point. However, in practice you usually have multiple data points to train on. Let's say we have N training data points. The most easist way to think is to feed the network with one data point at a time and repeat the forward and backward propagation N times. This method is called the online training. However, due to the noise in the data, this method lacks stability during the training phase.

Another updating method is called the (full) batch training. This method does not update weights and biases until all the data have made forward pass in the network. The error is accumulated for all training data points. After forward passes are made, the accumulated error is back propagated and the weight and biases are updated. In this method, error gradients w.r.t weights and biases are made by summing up the gradients obtained from each training data points.

$$\frac{\partial Loss(\mathbf{W}, \mathbf{X})}{\partial \mathbf{W}} = \sum_n \frac{\partial Loss(\mathbf{W}, \mathbf{x}_n)}{\partial \mathbf{W}}$$

In summary:

1. Forward propagate all the data points at once, while saving the input values of each layer $\mathbf{X}_l$. $\mathbf{X}_l$ will be a matrix of size $[N \times S_l]$ (S_l : the number of input nodes, N the number of training data points).

2. Calculate the error of the last layer δ_L, with has size of $[N \times S_L]$ (S_L : the number of output layer nodes).

3. Update weights, biases, and deltas (or delta errors) using the matrix form shown in the manual.

2.4 What to Do

You must fill in the portion of **neuralNetwork.py** during the assignment. You will fill in the following functions for this assignment:

- *forwardPropagation*
- *backwardPropagation*

In *forwardPropagation*, you should calculate the output of the neural network by doing forward propagation of the neural network. Please note that you also should store value of nodes of each layers' input to be used during the back propagation phase. You can **add any class variable** to store this information.

In *backwardPropagation*, you should calculate each layer's error (delta in the section 2.2) and the gradient of weights and biases using Eq. (13), Eq. (14), Eq. (16). You **don't need** to do l_2 regularization in this homework.

2.4.1 Network Structure

- Number of hidden layers: 2
- Number of units in each hidden layers: 100
- Activation function of the hidden layer: ReLU
- Acitvation function of the output layer:
  - For multi-class classification (hand-written): softmax
  - For binary classification (faces): sigmoid

Please refer Section 2.5 for more information.

2.5 Some Hints

- You can check the negative log likelihood value to ensure that your graident implementation is right. For sufficiently small learning rate, the cost should be decreased every epoch.

- Note that gradient descent method requires the learning rate that is 'small enough'. If the learning rate is too big, the gradient desecent 'overshoot' the minimum point and will be diverged. (https://wingshore.wordpress.com regression-in-one-variable-gradient-descent-contd/)

- In this homework, activation functions such as sigmoid, softmax, and ReLU function were introduced to simplify the neural net implementation. Also, I implemented loss functions for debugging purpose, such as binary cross-entropy and categorical cross-entropy. They are automatically selected by viewing the number of labels to be calssified, and saved as 'self.loss'. You may see the usage explained in a comment in the train function. Please use them freely.

- I also prepared several useful variables that you can use while implementing your code, including 'self.loss' which I mentioned earlier. Here is the list of prepared variables that you can use:

  - self.loss: (automatically selected) loss function
  - self.outAct: (automatically selected) output activation function
  - self.hiddenUnits: number of units in hidden layers
  - self.layerStructure: number of units in (hidden + output) layers
  - self.W: list of weights for each layers
  - self.b: list of biases for each layers
  - self.nLayers: number of (hidden + output) layers
    * Notice that len(self.W) = len(self.b) = self.nLayers
  - softmax, sigmoid, ReLU: useful functions for activation functinos. You may need ReLU only because others are combined as self.outAct.
  - self.epoch: number of gradient descent iteration.

- As for the recognition of faces that involves using sigmoid functions at the output, the result can be unstable for several reasons. Therefore, if your model have not achieved the accuracies written in pg.8, I will compare the output of each function.
- Please note that this homework will not have any base score, so do not submit the raw skeloton code, which will be useless.
- Program run time **should not exceed 10 minute**, for 2000 hand-written digit dataset and 450 face dataset.
- For testing purpose: on the handwritten image dataset, with 450 data, neural network shows 80% accuracy on validation set and 81% on test set. For 1500 data, neural network shows 88% accuracy on validation set and 87% on test set.
- On the face dataset that has 450 data, neural network shows 90% accuracy on validation set and 89% on test set.
- DO NOT use any other scientific library that are not allowed (e.g., scikit-learn).

2.6 What to Submit

Please submit **neuralNetwork.py** file **only**. Any late submissions will not be accepted.

2.7 How to Run the Code

To try out the classification pipeline, run **dataClassifier.py** from the command line. This will classify the digit data using the default classifier (mostFrequent) which blindly classifies every example with the most frequent label.

```
python dataClassifier.py
```

To activate the neural network classifier, use -c neuralNetwork, or -c nn:

```
python dataClassifier.py -c nn
```

To run on the face recognition dataset with different training data size, use -d faces and -t numTraining

```
python dataClassifier.py -c nn -d faces
```

```
python dataClassifier.py -c nn -t 1000
```