HW4. Neural Network*

Due: December 10, 2016

1 Introduction

In this assignment, you will design the neural network classifier on two image data sets, handwritten digit images and face images, which are used before.

2 Project Instruction

2.1 NumPy and Anaconda

In this assignment, we will use NumPy, but not SciPy, as the same as HW 1, 2, 3. Please notice that this homework **DOESN'T USE SciPy**. Also, we will use **Python 2.7** as the same as before. Programs for this homework:

- Python 2.7.11: https://www.python.org/
- NumPy 1.10.4: http://www.numpy.org/
- (Recommended) Anaconda: https://www.continuum.io/downloads/

2.2 Neural Network

In this homework, we will implement fully connected feed-forward neural network. Training the neural network can be divided into two parts: forward propgation and back propagation. In forward propagation phase, from given input data, each values of hidden nodes are calculated and at the end, values of the output nodes are generated as a output of the neural network. By the given training data, you can get the "error" of the neural network output, which is the difference from true output values. In the back propagation phase, the gradients of each neural network's parameters are calculated to minimize the error of the network's output.

2.2.1 Forward Propagataion

A neural network is composed with several hidden units and layers. There are three types of layers: input layer, hidden layer, and output layer. Each layers contains severl number of units (denoted as circle in the Fig. 1), and in this simple feed-forward neural network structure, units in one hidden layer are only affected by units in just previous layer. Notice that they are independent to the next layer's units, previous layers' units which are not adjacent, or even units within layer itself. Each units have its (real) value. From begining of the forward propagation, the values of the input layer units are assigned to the input data. The process of the forward propagation is, from the input layer, calculate the value of next layer's units one by one, and eventually get the value of output layer's unit which is the output value of this neural network.

*This programming assignment is originally made by Jungpyo Hong (jphong@ai.kaist.ac.kr)

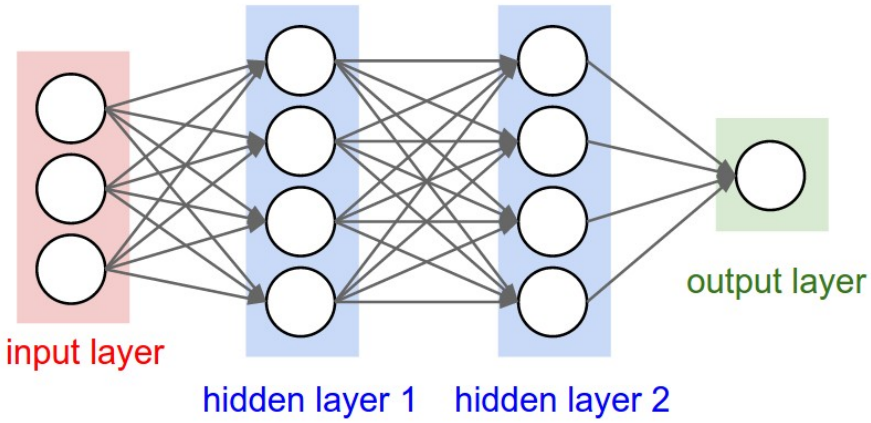

Figure 1: Neural network structure

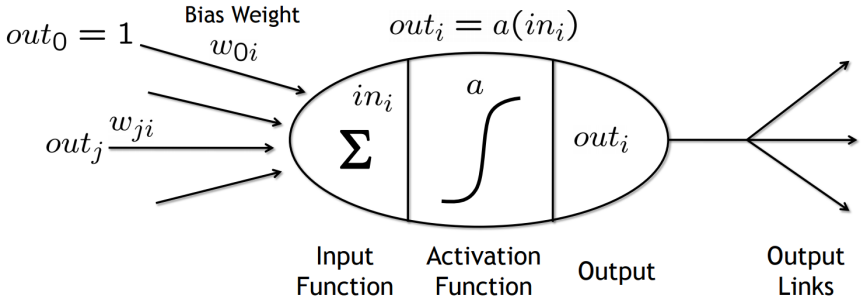

Figure 2: One node of an hidden layer

Let's look at one node of the hidden layer closely. The only value which affects the value of this node is the values the previous layer's units. Let the current hidden layer is l, and let the input of this hidden layer is $\mathbf{x}_l$. (So $\mathbf{x}_1$ is the input of the first hidden layer, which is the input data.) Our objective is to calculate one node's value of this layer, y_{lj} ($\mathbf{y}_l$ is the output of this layer, which is vector).

Each units of the hidden layer (and output layer) have its weights, bias and the activation function a. Let the weight and bias of this unit to $\mathbf{w}_{lj}$ and b_{lj}. We need to apply the weights and activation function to calculate y_{lj}. Applying weights is done by following:

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

Applying activation function is done by following:

$$y_{lj} = a(z_{lj})$$

Easy, right? Because NumPy likes the matrix form, we can combine all nodes of layer l to make this a matrix form. Let the weight and bias of this layer to $\mathbf{W}_l$ and $\mathbf{b}_l$. Please note that because now we combine vector $\mathbf{w}_{lj}$ and scalar b_{lj} into one for unit index j, they become matrix $\mathbf{W}_l$ and vector $\mathbf{b}_l$.

Applying weights is done by following:

$$\mathbf{z}_l = \mathbf{W}_l^\top \mathbf{x}_l + \mathbf{b}_l \tag{2}$$

(It may be more easier to calculate $\mathbf{Z}_l = \mathbf{X}_l^\top \mathbf{W}_l + \mathbf{b}_l$ in NumPy, if you are dealing with the several input data at once so $\mathbf{X}$ is now a matrix.)

Applying activation function is done by following:

$$\mathbf{y}_l = a(\mathbf{z}_l) \tag{3}$$

One may be notice that this form is very similar to the logistic regression. If the activation function is a softmax function, and there is no hidden layer:

$$\mathbf{y} = \text{Softmax}(\mathbf{W}^\top \mathbf{x} + \mathbf{b})$$

The value of current layer's unit is the input of next layer. So, $\mathbf{y}_l$ is the same as $\mathbf{x}_{l+1}$.

$$\mathbf{x}_{l+1} = \mathbf{y}_l$$

If there are L hidden and output layers, $\mathbf{y}_L$ is the output of the neural network.

2.2.2 Activation Functions

Activation functions, which is used to calculate the value of each units, is (normally) a non-linear function to give non-linearity of the neural network. In this homework, we uses three activation functions: ReLU, softmax, and sigmoid.

Rectified Linear Unit (ReLU)

ReLU activation function becomes very popular activation function nowadays for several reasons. The form of ReLU is like:

$$a(z) = \max(0, z)$$

which means makes the function value to 0 if the input is negative, or linear if it is postivie. ReLU activation function is known to accelerate the convergence, because it has higher gradient value on positive side compared to the other method. Also, it is very easy to compute. All units in the hidden layers in this homework will use ReLU, but for the output layer, we will use different activation functions.

Softmax

Softmax is the familiar function and you implemented it on HW3. The form of softmax is like:

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

Softmax activation function will be used for muti-class classification problem, and the reason to use is the same with HW3: we will model the output of neural network as the probability of class label, so summing them up will be 1. Also note that softmax function is not an element-wise operator, so output $\mathbf{y}_j$ is dependent on all elements of $\mathbf{z}$, not the $\mathbf{z}_j$ only.

Sigmoid

Sigmoid is usally used to the activation function of the output layer when dealing with the binary classification problem. The form of sigmoid is like:

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

Sigmoid function limits the value of units from 0 to 1. Because of this property, when we model the neural network for a binary classification problem, we usally set the number of unit in the output layer as one, set its activation function as the sigmoid function, and let its output as a probability that the input data is class 1. To get the probability of the other class, we subtract it from 1: $1 - y_L = 1 - \Pr(\text{label} = 1|x) = \Pr(\text{label} = 2|x)$

Linear

Linear function is the most simplest activation function:

$$a(z) = z$$

...Actually this is the same not to use the activation function. This function is normally used for the output layer when the neural network is modeled to solve the regression problem, to clarify that the neural network can represent any real value target data. Also, you can notice that if there is no hidden layer, and linear function is used for the output layer, the structure of the neural network becomes the same as the linear regression.

2.2.3 Back Propagation

Using several activation functions, and matrix multiplication thanks to NumPy, you may calculate the output of the neural network $\mathbf{y}_L$. If you want to train your neural network, you must calculate "error" from it, and update the weights and biases of each layers. Error in the neural network is defined as a loss function here. Let's assume we have $\mathbf{y}$ as a training target data (or y for scalar target data). Loss function become different between multi-class and binary classification problem. For binary classification, we use binary cross-entropy loss:

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

Here, y will be 0 if the data is "class 1" and 1 if the data is "class 2".

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

$$\mathrm{Loss}(\mathbf{y}, \mathbf{y}_L) = -\left[\sum_j \mathbf{y}_j \log(y_{Lj})\right]$$

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

To train your neural network, all weights and biases should move to where the loss function decreases. To find the direction of that way, you should calcualte the gradients of the parameters. Back propagation is an algorithm to calculate it in an efficient way.

Let's first think about the gradient of 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:

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

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:

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

For categorical cross-entropy loss:

$$\frac{\partial \mathrm{Loss}}{\partial y_{Lj}} = -\frac{\mathbf{y}_j}{y_{Lj}}$$

The second term is the derivation of each activation function, by Eq. (3). Graidents of linear and ReLU activation function is omitted. Gradient of the sigmoid function is:

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

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

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

For the softmax function, the chain rule becomes:

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

Gradient of the softmax function is:

$$\frac{\partial y_{Lj}}{\partial z_{Lj}} = y_{Lj}(1 - y_{Lj})$$

$$\frac{\partial y_{Lj}}{\partial z_{Lk}} = -y_{Lj} y_{Lk} \qquad \text{If } j \neq k$$

You can easily calculate the third term of the chain rule, $\frac{\partial z_{Lj}}{\partial \mathbf{W}_L}$ by Eq. (1)

$$\frac{\partial z_{Lj}}{\partial \mathbf{w}_{Lj}} = \mathbf{x}_l$$

$$\frac{\partial z_{Lj}}{\partial \mathbf{w}_{Lk}} = 0 \qquad \text{If } j \neq k$$

You can get the gradient of bias by very similar way (actually only the third term is changed).
Finally you must calculate the gradient about input of the last layer, $\mathbf{x}_L$, to propagating back.

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

$$\frac{\partial \text{Loss}}{\partial \mathbf{x}_L} = \frac{\partial \text{Loss}}{\partial \mathbf{y}_{L-1}} = \sum_j \frac{\partial \text{Loss}}{\partial y_{Lj}} \frac{\partial y_{Lj}}{\partial z_{Lj}} \frac{\partial z_{Lj}}{\partial \mathbf{x}_L} \tag{4}$$

For weight of layer $L-1$, $\mathbf{W}_{L-1}$,

$$\frac{\partial \text{Loss}}{\partial \mathbf{W}_{L-1}} = \sum_j \frac{\partial \text{Loss}}{\partial y_{(L-1)j}} \frac{\partial y_{(L-1)j}}{\partial z_{(L-1)j}} \frac{\partial z_{(L-1)j}}{\partial \mathbf{W}_{L-1}}$$

And you can notice that the first term is already calculated by Eq. (4). You can back propagate the gradient by calcualting gradient of input units for each layers one by one, like this way.

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:

$$\frac{\partial \text{Loss}}{\partial \mathbf{W}_l} = \mathbf{x}_l \delta_l^\top \tag{5}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{b}_l} = \delta_l \tag{6}$$

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,

$$\delta_{(l-1)k} = a'(z_{(l-1)k}) \sum_j \delta_{lj} w_{ljk} \tag{7}$$

$$\delta_{(l-1)} = \mathbf{W}_l \delta_l \circ a'(\mathbf{z}_{(l-1)}) \tag{8}$$

Here, $\circ$ is a element-wise multiplication.
Also, for ReLU, a' can be defined as $\mathbf{x}_l$, so

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

So, if you save all value of nodes $\mathbf{x}_l$, and can calculate δ_L, weights and biases can be updated like Eq. (5) and Eq. (6), and delta can be updated like Eq. (8)

How about δ_L? By calculating $\frac{\partial \text{Loss}}{\partial y_{Lj}} \frac{\partial y_{Lj}}{\partial z_{Lj}}$ by hand, you can derive that

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

for the cases (binary cross-entropy + sigmoid) and (categorical cross-entropy + softmax) (and l_2 loss + linear, which is explained on lecture slide).

2.3 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*

To simplify the work on this homework, implementing the validation of the model parameters are removed in HW6. Also, *forwardPropagation* will be used to calculate the conditional probability of the neural network.

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

In *backwardPropagation*, you should calculate each layers' error (delta in the section 2.2) and the gradient of weights and biases using Eq. (5), Eq. (6), Eq. (8). You **don't need** to do l_2 regularization, even if the training of neural networks normally uses the regularization.

2.3.1 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 2.4 for more information.

2.4 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 (in train method)
- Note that gradient descent method claims the the learning rate should be 'small enough'. If the learning rate is too big, the gradient desecent 'overshoot' the minimum point and will be diverged. (https://wingshore.wordpress.com/2014/11/19/linear-regression-in-one-variable-gradient-descent-contd/)
- In this homework, I delivered sigmoid, softmax, and ReLU function for simplify the 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 as a comment in the train function. Please use them freely.
- Section 2.2 usally talks about the training process using only one training data, but actually you may have many training data at once (as a type of NumPy array). I included the matrix form of the updates between the explanation, so please refer them. You can notice that processing several data at once is the same as processing one data at a time, but not updating the weights, and after processing all data then update weights with sum of the calculated gradients. So,

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

- I also prepared several useful variables that you can use while implementation, including 'self.loss' which I mentioned. Here is the list of prepared variable to 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 a logistic regression case, the result of this neural network will be unstable for several reasons. Therefore, I will **NOT** score your homework to how exact value you make on the accuracy. Rather than, I will compare the output of each function itself. Also please not that this homework will not have any base score, so do not submit the raw skeloton code, which will be useless.
- Also, your running time must **not be exceed to 10 minute**, for 2000 hand-written digit dataset and 450 face dataset, for flawless grading.
- For testing purpose: on the handwritten image dataset, with 450 data, neural network shows 84% accuracy on validation set and 81% on test set. For 1500 data, neural network shows 86% accuracy on validation set and 82% on test set.
- On the face dataset, with 450 data, neural network shows 90% accuracy on validation set and 87% on test set.
- DO NOT use any scientific library that abbreviate your implementation(i.e. scikit-learn)

2.5 What to Submit

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

2.6 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
```