Hw3 Perceptrons

Report written by Xiaodong Huo
March 20, 2020

part1:

1. Perceptrons is a binary linear classification algorithm that produces a linear decision boundary.

2. The C-class classification is similar to the binary case.

$$\hat{y} = argmax_y f(x).w_y$$

Start with weight 0, we loop through all the training examples one by one, make a prediction with the current weight vector we have. The prediction will be argmax of all the labels of the dot product of the weight vector and feature vector. We have multiple weight vectors, and we are going to update the weight vector with error. We then lower the score of the wrong answer and increase the score of the correct answer. For example, if the correct label of the feature vector $f(x)$ is y_c. We then first subtract f(x) from the incorrect weight vector, and add f(x) to the correct weight vector. $w_y = w_y - f(x)$ $w_{yc} = w_{yc} + f(x)$. Thus, the correct weight vector will approach to f(x) and the incorrect weight vector will deviate from f(x). This is done for all training examples.

3. Gradient Descent process is we first have a cost function, the cost function measures the difference between the true value and predicted value. The following is an example of a cost function:

$$J(\theta_0, \theta_1) = \frac{1}{2m}\Sigma_m[h_\theta(x_i) - y_i]^2$$

the gradient Descent formulation then becomes:

$$\theta_j = \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta_0, \theta_1)$$

Where α is the learning rate. The standard gradient descent calculates all samples for each step, where Stochastic Gradient Descent calculates one random sample each step. Thus, Stochastic Gradient Descent works well for big data compares with Gradient Descent. First, stochastic gradient decent randomly picks one sample for each step, and use just one sample to calculate derivatives instead of using all the training samples. For example, if we have 10 million samples, stochastic gradient descent would reduce the amount of terms by 10million.

4. We can first vectorize all the training examples, let's say we have k examples:

$$X = [x^1, x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9, ...x^k]$$

The dimension of X would be (n_x, k). We need to process the entire training set before we take a gradient descend step. Thus, if we have a large k, it would take a long time. Mini-batch gradient decent solves this problem. For example each mini-batch have 3 examples. X is segmented into

$$X^{[1]} = x^1, x^2, x^3, X^{[2]} = x^4, x^5, x^6, X^{[3]} = x^7, x^8, x^9...$$

the size of $X^{[1]}, X^{[2]}, X^{[3]}$ have size of $(n_x, 3)$. Rather than processing the entire dataset, we can process mini-batch data-set instead. Then, we have the same process as gradient descend. The difference is the training examples are mini-batchs. We don't need to loop through each element in the batch, we can just use vector calculation to speed up the process.

5.

$$\sigma' = \frac{e^{(-a)}}{(1+e^{(-a)})^2}$$

$$\sigma' = \frac{1+e^{(-a)}}{(1+e^{(-a)})^2} - \frac{1}{(1+e^{(-a)})^2}$$

$$\sigma' = \frac{1}{(1+e^{(-a)})} - \frac{1}{(1+e^{(-a)})^2}$$

$$= \frac{1}{(1+e^{(-a)})}(1 - \frac{1}{(1+e^{(-a)})}) = \sigma(a) * (1 - \sigma(a))$$

6. One Epoch is a hyperparameter equals to 1 forward pass + 1 backward pass for an entire dataset through the Neural network once.

7. 2000/50 = 40 batchs. We need 40 iterations to complete one Epoch.

8. Dropout is the process that randomly drop the nodes in each layer of the network with a probability. So that we can reduce overfitting. After dropout for one sample, we then do backpropagation training this one example. We dropout different nodes for different training examples.

9. Early stopping is a regularization technique that reduces overfitting when we do iterative learning such as gradient decent. There are several ways to do early stopping: 1. To train only a preset number of Epochs. 2. Stop when updates are sufficiently small.3. Validation based early stopping. As we perform training, at first, the loss function decrease exponentially, and then it approaches flat. The validation loss however, decrease exponentially at first, and then goes up as time increases. As soon as validation error and training error diverges, we should stop the training.

10. Regularization reduces overfitting. Overfitting happens when the function trained itself to trying to fit every data exactly in the training examples so the pattern is lost. Regularization reduces overfitting by adding an additional penalty term in the error function so that the function doesn't take extreme coefficients.

11. Early stopping; Data augmentation; Regularization; Dropouts; Change the model; those are ways to reduce overfitting in a network.

12. Batch normalization improves the stability, speed and performance of the network. It improves the speed because it makes the optimization much smoother. It decreases the importance of initial weight and regularize the model. We have a mini batch of m samples; form $B = [x_1, ...x_m]$. The activation mean is $\mu_B = \frac{1}{m}\Sigma_m(x_i - \mu_B)^2$, variance is $\sigma_B^2 \frac{1}{m}\Sigma_m(x_i - \mu_B)^2$. We then normalize by

$$x_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

we finally have two learning parameters γ and β. Where

$$y_i = \gamma x_i + \beta$$

Part 2. Perceptrons: Initialization of the data. In the loading data function, one of the most important functionality is to convert the label to one hot label to prepare for the upcoming test. For instance, if the label is 2, I am converting it to [0,1,0,0,0,0,0,0,0,0]. For the sake of saving time, the data is stored as pickle data for fast loading.

Question1; I developed a simple NN with 10 perceptrons. Each have 784 input and a output. I used early stopping techniques when training to avoid overffiting. The gradient descent rate is 0.1. The weight is initialized with Gaussian. By observation, this simple network don't have a really high performance.

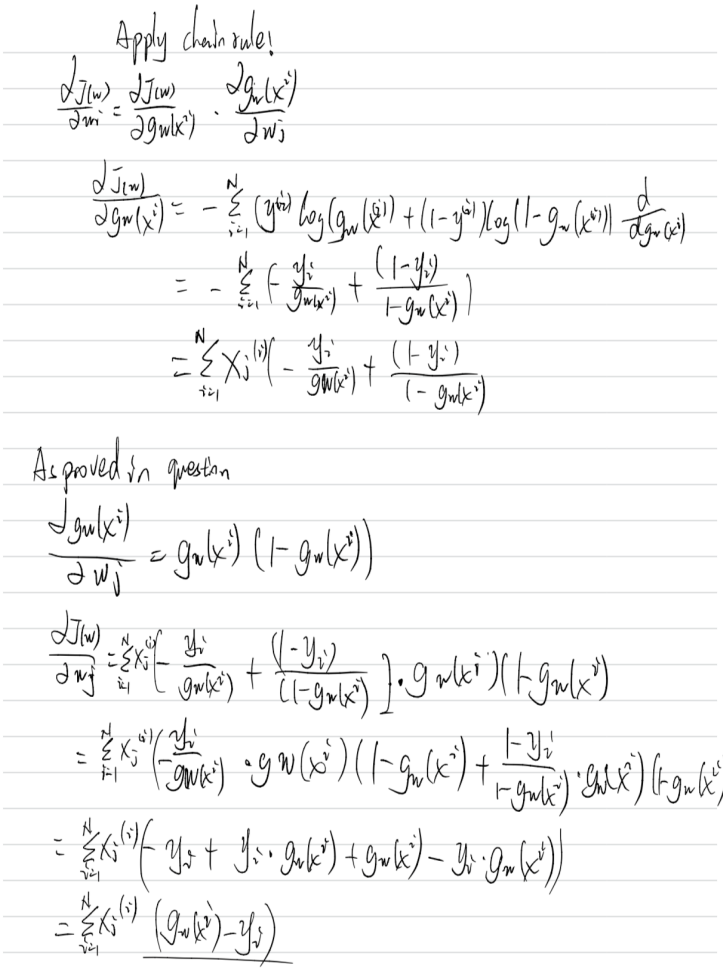
  - The overall error rate is 19.2 %

- The overall accuracy is 80.8 %
- The error rate for number 0 is 7.741935483870968 %
- The error rate for number 1 is 10.980392156862745 %
- The error rate for number 2 is 9.090909090909092 %
- The error rate for number 3 is 11.176470588235295 %
- The error rate for number 4 is 29.491525423728817 %
- The error rate for number 5 is 35.146443514644346 %
- The error rate for number 6 is 16.666666666666664 %
- The error rate for number 7 is 17.277486910994764 %
- The error rate for number 8 is 18.75 %
- The error rate for number 9 is 25.257731958762886 %

Updates the weights:

$$w_i = w_i(t) + r(d_j - y_j(t))x_{j,i}$$

Part 3. Logistic Regression:



  - **Question 23 the individual accuracy and the total accuracy.**

This network shares a lot of similarities with the previous network. I design a network with 1 hidden layer which has 500 hidden nodes. 1 input layer with 10 nodes and each node have a 784 input which correspond to each pixel of the image. I used the sigmoid function for my activation function. Initialize the weight parameters with Gaussian distribution, and used the logistic regression function for weight updates.

- The overall error rate is 6.4 %
- The overall accuracy is 93.6 %
- The error rate for number 0 is 4.4692737430167595%
- The error rate for number 1 is 3.3472803347280333%
- The error rate for number 2 is 4.225352112676056%
- The error rate for number 3 is 2.7027027027027026%
- The error rate for number 4 is 3.8461538461538463%
- The error rate for number 5 is 8.64864864864865%
- The error rate for number 6 is 7.065217391304348%
- The error rate for number 7 is 8.374384236453201%
- The error rate for number 8 is 11.961722488038278%
- The error rate for number 9 is 9.743589743589745%

We can see from the results, the number 8 has the lowest accuracy out of all the digits. This network using logistic regression improves the accuracy by a large margin.

References