Handin 2: Neural Nets for Multiclass Classification

Alexia Borchgrevink (201801317)
Marwan Bushara (201801608)
Shunya Kogure (201800953)

October 29, 2018

1 Derivative

Based on our problem and given that the Softmax function is:

$$S_j = \frac{e^{z_j}}{\sum_{t=1}^{k} e^{z_t}}, \forall j \in 1, ..., k \tag{1}$$

In order to find its derivative we need to look for its partial derivatives. Therefore we solve for:

$$\frac{\partial S_i}{\partial z_j} = \frac{\partial \frac{e^{z_i}}{\sum_{t=1}^{k} e^{z_t}}}{\partial z_j} \tag{2}$$

Which translates to "the partial derivative of the i-th output with respect to the j-th input" which in our case is vector z of dimension k. We then use the quotient rule and separate our operation into two different cases since the derivative of e^{z_j} is e^{z_j} only when i=j, otherwise it is 0. Let S be the softmax function with input z.

Case: i = j

$$\begin{aligned} \frac{\partial S_i}{\partial z_j} &= \frac{e^{z_i} \sum_{t=1}^{k} e^{z_t} - e^{z_j} e^{z_i}}{(\sum_{t=1}^{k} e^{z_t})^2} \\ &= \frac{e^{z_i}}{\sum_{t=1}^{k} e^{z_t}} \frac{\sum_{t=1}^{k} e^{z_t} - e^{z_j}}{\sum_{t=1}^{k} e^{z_t}} \end{aligned} \tag{3}$$

Which can be simplified to:

$$S_i(1 - S_j) \tag{4}$$

Case: i ≠ j

Similarly, when i $\neq$ j, we have that

$$\begin{aligned} \frac{\partial S_i}{\partial z_j} &= \frac{0 - e^{z_j} e^{z_i}}{(\sum_{t=1}^{k} e^{z_t})^2} \\ &= -\frac{e^{z_j}}{\sum_{t=1}^{k} e^{z_t}} \frac{e^{z_i}}{(\sum_{t=1}^{k} e^{z_t})^2} \end{aligned} \tag{5}$$

Which simplifies to:

$$-S_j S_i \tag{6}$$

Using equations (4) and (6), the derivative for the softmax function with input z summarizes in:

$$\frac{\partial S_i}{\partial z_j} = \begin{cases} S_i(1 - S_j) & \text{if i = j} \\ -S_j S_i & \text{if i} \neq \text{j} \end{cases} \tag{7}$$

If we use the Kronecker Delta function [1] we would have:

$$\frac{\partial S_i}{\partial z_j} = S_i(\delta_{ij} - S_j) \tag{8}$$

Now that we have the derivation of the softmax function, we are going to use the chain rule to obtain $\frac{\partial L}{\partial z_i}$ as follows:

$$\begin{aligned} \frac{\partial L}{\partial z_i} = \frac{-1}{S_j} * S' \\ = \frac{-1}{S_j} * S_j(\delta_{ij} - S_i) \\ = \frac{-S_j(\delta_{ij} - S_i)}{S_j} \\ = -\delta_{ij} + S_i \end{aligned} \tag{9}$$

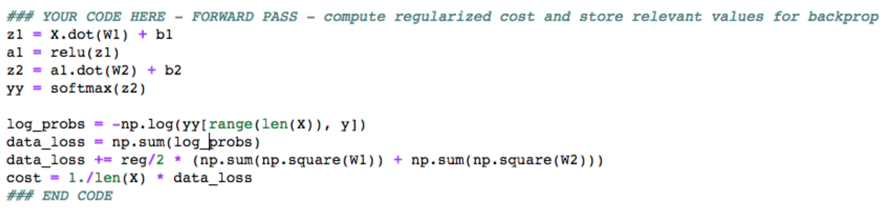
Which is the same as: $-\delta_{ij} + softmax(z)_i$

2 Implementation and Test

2.1 Code

2.1.1 Forward Pass

Since the neural network we implemented had three layers in total, we had two weights and two biases to calculate with the data inputs. Below is the code snippet, where we implemented the forward pass.



```
### YOUR CODE HERE - FORWARD PASS - compute regularized cost and store relevant values for backprop
z1 = X.dot(W1) + b1
a1 = relu(z1)
z2 = a1.dot(W2) + b2
yy = softmax(z2)

log_probs = -np.log(yy[range(len(X)), y])
data_loss = np.sum(log_probs)
data_loss += reg/2 * (np.sum(np.square(W1)) + np.sum(np.square(W2)))
cost = 1./len(X) * data_loss
### END CODE
```

Figure 1: Code Snippet for forward pass

z1 is the product sum of the product between data point and each weight that corresponds, also the bias will be added. a1 is the output of relu function that

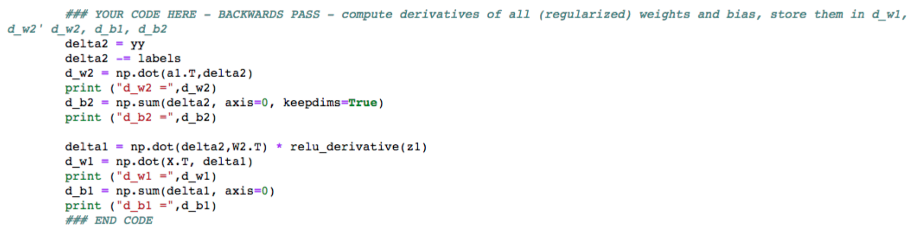
[1]Weisstein, Eric W. "Kronecker Delta." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/KroneckerDelta.html

has z1 as the input this is our hidden layer. Relu function acts as the activation function of the neuron. Then, a1 will be multiplied by each corresponding weights for the hidden layer plus again, the bias. The output of this calculation (z2) will be input to the softmax function which will be the output layer of our neural network.

We had some issues with the cost since as we ran net_test.py, all the results we got were NaN and we also had issues with numerical_grad_check.

2.1.2 Backward Pass

As for the backward pass, we have the following code:



```
        ### YOUR CODE HERE - BACKWARDS PASS - compute derivatives of all (regularized) weights and bias, store them in d_w1,
d_w2' d_w2, d_b1, d_b2
        delta2 = yy
        delta2 -= labels
        d_w2 = np.dot(a1.T,delta2)
        print ("d_w2 =",d_w2)
        d_b2 = np.sum(delta2, axis=0, keepdims=True)
        print ("d_b2 =",d_b2)

        delta1 = np.dot(delta2,W2.T) * relu_derivative(z1)
        d_w1 = np.dot(X.T, delta1)
        print ("d_w1 =",d_w1)
        d_b1 = np.sum(delta1, axis=0)
        print ("d_b1 =",d_b1)
        ### END CODE
```

Figure 2: Code Snippet for backward pass

The backpropagation algorithm obtains the output of the feedforward and starts computing the derivatives of the output for each layer with respect to weights and biases in each layer. These will be the values that will update the weights and biases when we run the mini batch SGD in the fit function.

Mainly because of time constraints, we were unable to solve the issues we had and generate the graph. However, the most important part was understanding the backpropagation algorithm which we consider the exercise allowed us to understand it very well.

References

[1] Abu Mostafa, Y. S., Magdon-Ismail, M., and Lin, H. (2012). *Learning from data: A short course.*. S. l.: AML.