Ryan George
Deep Learning HW2

Problem 1

1. Provide a geometric interpretation of gradient descent in the one-dimensional case. (Adapted from the Nielsen book, chapter 1)

> Gradient descent in one-dimension measures only two things: the direction and magnitude of the slope at the current place. If the slope is positive, the descent will be to the left (in the negative direction). If the slope is negative, the descent will be to the right (in the positive direction). The greater the magnitude of the slope, the further the descent to the given side.

2. An extreme version of gradient descent is to use a mini-batch size of just 1. This procedure is known as online or incremental learning. In online learning, a neural network learns from just one training input at a time (just as human beings do). Name one advantage and one disadvantage of online learning compared to stochastic gradient descent with a mini-batch size of, say, 20. (Adapted from the Nielsen book, chapter 1)

> One advantage of online learning is that each example benefits from anything learned from previous examples. In that sense, bias is decreased for each given example. The disadvantage of online learning is that there is greater noise each time the weights are updated. When the mini-batch is larger than 1, the “errors in the errors” are averaged away, but this can’t happen with online learning.

3. Create a network that classifies the MNIST data set using only 2 layers: the input layer (784 neurons) and the output layer (10 neurons). Train the network using stochastic gradient descent. What accuracy do you achieve? You can adapt the code from the Nielson book, but make sure you understand each step to build up the network. Please save your code as prob1.py. (Adapted from the Nielsen book, chapter 1)

```
>>> nw.SGD(training_data, 30, 10, 3.0, test_data=test_data)
Epoch 0: 8015 / 10000
Epoch 1: 8152 / 10000
....
Epoch 17: 9163 / 10000
Epoch 19: 9171 / 10000
Epoch 20: 9154 / 10000
```

We get only to 92% accuracy. For another experiment, we try online learning:

```
>>> nw.SGD(training_data, 30, 1, 3.0, test_data=test_data)
```

```
Epoch 0: 8781 / 10000
Epoch 1: 8958 / 10000
...
Epoch 28: 9088 / 10000
Epoch 29: 9110 / 10000
```

Problem 2

1. Show that δL = ∇aC ⊙ σ′(zL) can be written as δL = ′(zL)∇aC, where Σ′(zL) is a square matrix whose diagonal entries are the values σ′(zjL) and whose off-diagonal entries are zero.

Suppose $\delta^L = \nabla_a C \odot \sigma'(z^L)$. So we can express the ith element of $\delta^L = \delta^L{}_i$ as

$$\delta^L{}_i = \nabla_a C_i \cdot \sigma'(z^L{}_i)$$

Now suppose we have the matrix $\Sigma'(z^L)$ whose diagonal entries are the values $\sigma'(z^L{}_i)$ and whose off-diagonal entries are zero.

Then define $\zeta^L = \Sigma'(z^L)\ \nabla_a C$. We can see from matrix multiplication that the ith element of $\zeta_i{}^L = \sigma'(z^L) \cdot e_i\ \nabla_a C$ (where e_i is a vector with a value 1 in the ith index and zero elsewhere). Then:

$$\zeta_i{}^L = \sigma'(z^L) \cdot e_i\ \nabla_a C\ = \sigma'(z^L{}_i)\ \nabla_a C_i = \delta^L{}_i$$

Since i was arbitrary, $\zeta_i{}^L = \delta^L{}_i$, so $\nabla_a C \odot \sigma'(z^L) = \delta^L\ = \zeta^L = \Sigma'(z^L)\ \nabla_a C$

2. Show that δl = ((wl+1)T δl+1) ⊙ σ′(zl) can be rewritten as δl = Σ′(zl)(wl+1)T δl+1.

The ith element of $((w^{l+1})^T \delta^{l+1})$ is $((w^{l+1}{}_i)^T \delta^{l+1})$, where $w^{l+1}{}_i$ is the ith column of w^{l+1} .

So the ith element of $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot\ \sigma'(z^l)$ is:

$$\begin{aligned}\delta^l{}_i &= ((w^{l+1}{}_i)^T \delta^{l+1}) \odot\ \sigma'(z^l)_i \\ &= ((w^{l+1}{}_i)^T \delta^{l+1}) \odot\ \sigma'(z^l{}_i)\end{aligned}$$

Similarly to problem one, define $\zeta^l = \Sigma'(z^l)((w^{l+1})^T \delta^{l+1})$. Then per matrix multiplication, the the ith element of ζ^l is:

$$\begin{aligned}\zeta_i{}^l &= \sigma'(z^l) \cdot e_i \odot ((w^{l+1})^T \delta^{l+1}) \\ &= ((w^{l+1}{}_i)^T \delta^{l+1}) \odot\ \sigma'(z^l{}_i) \\ &= \delta^l{}_i\end{aligned}$$

So $\zeta^l = \delta^l$ and $\Sigma'(z^l)((w^{l+1})^T \delta^{l+1}) = \Sigma'(z^l)(w^{l+1})^T \delta^{l+1} = \zeta^l = \delta^l = ((w^{l+1})^T \delta^{l+1}) \odot\ \sigma'(z^l)$

3. By combining the results from problems 2.1 and 2.2, show that δl = Σ′(zl)(wl+1)T . . . Σ′(zL−1)(wL)T Σ′(zL)∇aC.

For layer L, $\delta^L = \Sigma'(z^L) \ \nabla_a C$

For layer $L-1$,

$$\delta^{L-1} = \Sigma'(z^{L-1})(w^L)^T \delta^L$$
$$= \Sigma'(z^{L-1})(w^L)^T \ \Sigma'(z^L)((w^L)^T \delta^L)$$

For layer l,

$$\delta^l = \Sigma'(z^l)(w^{l+1})^T \delta^{l+1}$$
$$= \Sigma'(z^l)(w^{l+1})^T \Sigma'(z^{l+1})(w^{l+2})^T \delta^{l+2}$$
$$\dots$$
$$= \Sigma'(z^l)(w^{l+1})^T \Sigma'(z^{l+1})(w^{l+2})^T \dots \Sigma'(z^{L-1})(w^L)^T \delta^L$$
$$= \Sigma'(z^l)(w^{l+1})^T \Sigma'(z^{l+1})(w^{l+2})^T \dots \Sigma'(z^{L-1})(w^L)^T \Sigma'(z^L) \ \nabla_a C$$

4. Suppose we replace the usual non-linear σ function (sigmoid) with σ(z) = z throughout the network. Rewrite the backpropagation algorithm for this case.

If $\sigma(z) = z$, then $\sigma'(z) \equiv 1$. In that case, $\Sigma'(z) = I$, so:

$$\delta^L = \Sigma'(z^L) \ \nabla_a C$$
$$= I \ \nabla_a C$$
$$= \nabla_a C$$

For other layers,

$$\delta^l = \Sigma'(z^l)(w^{l+1})^T \delta^{l+1}$$
$$= I(w^{l+1})^T \delta^{l+1}$$
$$= (w^{l+1})^T \delta^{l+1}$$
$$= (w^{l+1})^T (w^{l+2})^T \dots (w^L)^T \ \nabla_a C$$

The actual implementation of the algorithm then stays the same:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$$

Problem 3

1. It can be difficult at first to remember the respective roles of the ys and the as for cross-entropy. It's easy to get confused about whether the right form is −[ylna + (1 − y)ln(1 − a)] or −[alny + (1 − a)ln(1 − y)]. What happens to the second of these expressions when y=0 or 1? Does this problem afflict the first expression? Why or why not?

To frame the problem, recall that y are the desired outputs and a are the actual outputs. In a simple example, we might want $y_i \varepsilon \{0, 1\}$. In the second expression, we have:

$$\text{error} = -[a\, ln(y) + (1-a) ln(1-y)]$$

If $y = 1$,

$$\begin{aligned}\text{error} &= -[a\, ln(1) + (1-a) ln(1-1)] \\ &= -[a \cdot 0 + (1-a) ln(0)] \\ &= -[(1-a)(-\infty)] \\ &\sim = -\infty \text{ (not really well-defined)}\end{aligned}$$

And if $y = 0$,

$$\begin{aligned}\text{error} &= -[a\, ln(0) + (1-a) ln(1-0)] \\ &= -[a\,(-\infty) + (1-a) ln(1)] \\ &\sim = -\infty\end{aligned}$$

This doesn't affect the first equation. If y=1, then:

$$\begin{aligned}\text{error} &= -[1 \cdot ln(a) + (1-1) ln(1-a)] \\ &= -[ln(a) + 0 \cdot ln(1-a)] \\ &= -ln(a)\end{aligned}$$

And if y=0, then:

$$\begin{aligned}\text{error} &= -[0 \cdot ln(a) + (1-0) ln(1-a)] \\ &= -ln(1-a)\end{aligned}$$

Essentially, it's ok for our outputs to be completely certain, but not for our network to ever be completely certain.

2. Show that the cross-entropy is still minimized when σ(z) = y for all training inputs (i.e. even when y ∈ (0, 1)). When this is the case the cross-entropy has the value: C = − 1 [ylny + (1 − y)ln(1 − y)]

We have the cross-entropy cost function as:

$$C = -\frac{1}{n} \Sigma_x [y ln(a) + (1-y) ln(1-a)]$$

So the derivative with respect to $a = \sigma(z)$:

$$\begin{aligned}0 = \frac{\delta C}{\delta a} &= -\frac{1}{n} \Sigma_x [y \frac{1}{a} - (1-y) \frac{1}{1-a}] \\ \Rightarrow\ 0 &= \Sigma_x [y(1-a) - (1-y)a]\end{aligned}$$

This holds if, for each training example x, we have $y(1-a) = (1-y)a$.

If $y\varepsilon(0,1)$, then $a = y$ satisfies this, since:

$$y(1-a) = (1-y)a$$
$$\Rightarrow y(1-y) = y(1-y)$$

If $a = \sigma(z) = y$, then:

$$C = -\frac{1}{n}\Sigma_x[yln(a) + (1-y)ln(1-a)]$$
$$= -\frac{1}{n}\Sigma_x[yln(y) + (1-y)ln(1-y)]$$

3. Given the network in Figure 1, calculate the derivatives of the cost with respect to the weights and the biases and the backpropagation error equations (i.e. δl for each layer l) for the first iteration using the cross-entropy cost function.

We must first do a forward pass of the network. With initial input .05, and .1,
Value from h1 is $a_1{}^1 = \sigma(.15 * .05 + .1 * .25 + .35) = \sigma(.3825) = .5945$
Value from h2 is $a_2{}^1 = \sigma(.25 * .05 + .1 * .3 + .35) = \sigma(.3925) = .5969$

Value from o1 is $a_1{}^2 = \sigma(.5945 * .4 + .5969 * .45 + .6) = \sigma(1.106) = .7514$
Value from o2 is $a_2{}^2 = \sigma(.5945 * .5 + .5969 * .55 + .6) = \sigma(1.226) = .7731$

Now, from Nielson we have that:

$$\delta^L = a^L - y$$

So $\delta^2 = a^2 - y = (.7514 - .01, .7731 - .99) = (.7414, -.2169)$
Then $\delta^1 = ((w^2)^T\delta^2) \odot \sigma'(z^2) =$

$$\begin{pmatrix} .4 & .45 \\ .5 & .55 \end{pmatrix}\begin{pmatrix} .7414 \\ -.2169 \end{pmatrix} = \begin{pmatrix} 0.198955 \\ 0.251405 \end{pmatrix}$$

We compose this with $\sigma'(z^2) = \begin{pmatrix} 0.1868 \\ .1754 \end{pmatrix}$ to get $\delta_1 = (.0372, .0441)$
Then $\delta^0 = ((w^1)^T\delta^1) \odot \sigma'(z^1) =$

$$\begin{pmatrix} .15 & .25 \\ .2 & .3 \end{pmatrix}\begin{pmatrix} .0372 \\ .0441 \end{pmatrix} = \begin{pmatrix} 0.198955 \\ 0.251405 \end{pmatrix}$$

Composing this with $\sigma'(z^1) = (.2291, .2290)$,

$$\delta^0 = (.0456, .0576)$$

So $\frac{\delta C}{\delta b^1} = \delta_1 = (.0372, .0441)$

And $\frac{\delta C}{\delta w^1} = a^0\delta^1{}^T = (.5945, .5969)(.0372, .0441)^T = \begin{pmatrix} 0.0221154 & 0.02621745 \\ 0.02220468 & 0.02632329 \end{pmatrix}$
Also $\frac{\delta C}{\delta b^2} = \delta_2 = (.7414, -.2169)$

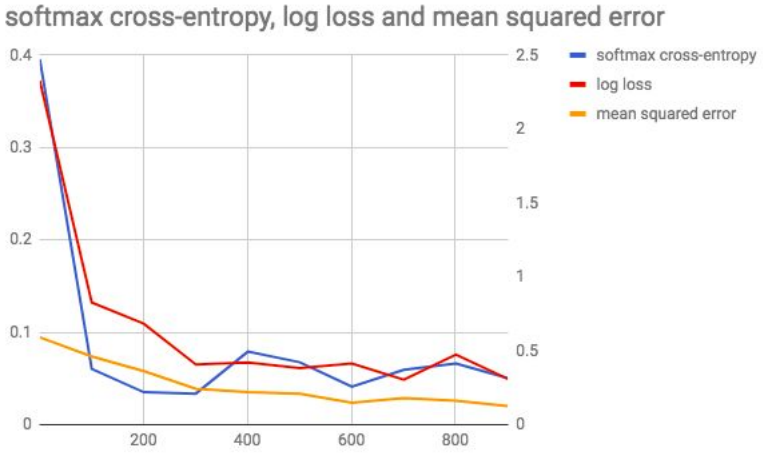
And $\frac{\delta C}{\delta w^2} = a^1\delta^2{}^T = (.7514, .7731)(.7414, -.2169)^T = \begin{pmatrix} 0.55708796 & -0.16297866 \\ 0.57317634 & -0.16768539 \end{pmatrix}$

Note: it was a little unclear whether w2 was from i1 to h2 or from i2 to h1 (etc.)

Problem 4

1. Download the python template prob4 1.py and read through the code which implements a neural network with TensorFlow based on MNIST data. Implement the TODO part to define the loss and optimizer. Compare the squared loss, cross entropy loss, and softmax with log-likelihood. Plot the training cost and the test accuracy vs epoch for each loss function (in two separate plots). Which loss function converges fastest?

The loss function evaluations converge at similar rates for cross entropy and log loss, but appear to converge yet a little faster with cross entropy:



This is also borne out in the final accuracy measures:

- Cross entropy: 0.9141
- Log loss: 0.9068
- Mean squared error: 0.8831

The relevant code is in prob4_1.py

2. Based on prob 4.1 add regularization to the previous network. Implement L2 and L1 regularizations separately, and dropout separately. Compare the accuracy and report the final regularization parameters you used (for dropout, report the probability parameter). Are the final results sensitive to each parameter?

The results for different parameters in l1 and l2 regularization, and for dropout, are:

reg param	l1 reg acc.	l2 reg acc.	keep probability	dropout acc.
0.001	89%	91%	20%	48%
0.005	85%	92%	50%	82%
0.01	11%	91%	80%	89%
1	NaN	11%	90%	91%

These are certainly sensitive to the parameters used.

Bonus

Where does the softmax name come from?

> The softmax function has a similar effect as setting the maximum value to 1 and all others to zero, but it does so in a less strong fashion. As we saw from question 3.1, the cross-entropy loss function doesn't want us to ever be completely certain of any network output (i.e. output should never be 0 or 1). Softmax allows us to emphasize the value of the "winning" neuron and diminish the values of others while keeping to this rule.