Sojeong Yang
CSC 578 Homework #5 **Backprop Hyper-Parameters**

1. Experiment results:

	act_hidden	act_output	cost	regularization	lmbda	dropout	result
1	Sigmoid	Sigmoid	Quadratic	(default)	0.0	0.0	match
2	Sigmoid	Sigmoid	CrossEntropy	(default)	0.0	0.0	match
3	Sigmoid	Softmax	CrossEntropy	(default)	0.0	0.0	match
4	Sigmoid	Softmax	LogLikelihood	(default)	0.0	0.0	match
5	ReLU	Softmax	CrossEntropy	(default)	0.0	0.0	match
6	ReLU	Softmax	LogLikelihood	(default)	0.0	0.0	match
7	Tanh	Sigmoid	Quadratic	(default)	0.0	0.0	match
8	Tanh	Tanh	Quadratic	(default)	0.0	0.0	match
9	Sigmoid	Sigmoid	Quadratic	L2	3.0	0.0	match
10	Sigmoid	Sigmoid	Quadratic	L1	3.0	0.0	match
11	Sigmoid	Sigmoid	Quadratic	(default)	0.0	0.1	Result may vary
12	Sigmoid	Sigmoid	Quadratic	(default)	0.0	0.5	Result may vary

According to the instruction of homework, the results of experiment 11 and 12 may vary because dropout nodes are randomly chosen. I think that my result is similar to the given sample-result, so my dropout process works fine.

2. Implementation

a) CrossEntropyCost derivative:

From the NNDL book, the cross-entropy cost function is $C = -\frac{1}{n}\sum_x[y \ln a + (1-y)\ln(1-a)]$ and it matched to the given function fn(a, y): `np.sum(np.nan_to_num(-y*np.log(a)-(1-y)* np.log(1-a)))`. If we use the derivative of the above cost function, we get this:

$\frac{\partial C}{\partial w_i} = \frac{\partial C}{\partial a} \cdot [\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i}]$ (using chain rule)

$= \frac{\partial}{\partial a}\{\frac{1}{n}\sum_x[-y(x)\ln a - (1-y)x)) \cdot ln(1-a)]\} \cdot [\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i}]$

$= \frac{1}{n}\sum_x -\{\frac{\partial}{\partial a}[y(x)\ln a] + \frac{\partial}{\partial a}[(1-y(x))\ \ln(1-a)]\} \cdot [\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i}]$

$= \frac{1}{n}\sum_x -\{\frac{y(x)}{a} + [(1-y(x))\ \frac{\partial}{\partial a}\ln(1-a)]\} \cdot [\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i}]$

$= \frac{1}{n}\sum_x -\{\frac{y(x)}{a} + \frac{(1-y(x))}{(1-a)} \cdot (-1)\} \cdot [\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i}]$

$= \frac{1}{n}\sum_x -\{\frac{a-y(x)}{a(1-a)}\} \cdot [\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i}]$

From the above result, I wrote a code:
`np.divide((a-y), (a*(1-a)), out=np.zeros_like(a), where=((a*(1-a))!=0))`
We need to think the case, a = 0 and a = 1. If a = 0 or a = 1, the dominant is going to be zero. The code works in case of a ≠ 0 or a ≠ 1 because it only divides the calculation where (a*(1-a)) does not equal to zero. So, we get (a-y)/(a*(1-a)) and the other case, we get zero.

b) LogLikelihood function, derivative:

The log-likelihood cost function is $C = \frac{1}{n}\sum_x -\ln(a_y^L)$. In the function, 'ln' means natural log so I use np.log to build the code.(np.log(np.e) = 1.0) In the formula, y is the corresponding desired output. If the network is doing a good job, it will estimate a value for the corresponding probability a_y^L which is close to 1. And the other components of y which is not the index of the target, are going to be zero. So, I write the code for 'def fn(a, y)': `-np.sum(np.nan_to_num(np.log(a [np.where(y==1)])))` To get the derivative of the log-likelihood cost function, I followed this:

$\frac{\partial C}{\partial w_i} = \frac{\partial C}{\partial a} \cdot [\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i}]$ (using chain rule)
$= \frac{\partial}{\partial a}\{\frac{1}{n}\sum_x[-\ln(a_y^L)]\} \cdot [\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i}]$
$= \frac{1}{n}\sum_x \frac{\partial}{\partial a}\{-\ln(a_y^L)\} \cdot [\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i}]$
$= \frac{1}{n}\sum_x\{-\frac{1}{a_y^L}\} \cdot [\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i}]$

c) Softmax function

The Softmax function almost only applied to the output layer and it was also used in the output layer of our experiments. From the NNDL book, the softmax function is: $a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$. From the function, we can see that all values are positive by taking the exponent. For each output node is going to be: $\frac{e^{net\ input}}{sum\left(=\sum_k e^{z_k^L}\right)}$. It we sum it up; it is going to 1 because the function makes them a probability distribution. Therefore, if the target output is close to 1, the others are close to zero. We are already given the derivative of softmax function, `a=cls.fn(z)` and `np.diagflat(a)-np.dot(a, a.T)`. To get this, we need to understand how it get an N*N matrix. The derivative of softmax is not a vector because the activation of a_j relies on all other activations ($\sum_k a_k$). (for each output node j, each layer has k nodes) np.diagflat(a) makes two-dimensional array with the flattened input as a diagonal. So, it makes us calculate this:

s_i(softmax) $= \frac{e^{z_i}}{\sum_k e^{z_k}}$, $\frac{\partial s_i}{\partial a_j} = \begin{cases} s_i(1-s_j), when\ i = j \\ -s_i \cdot s_j, when\ i \neq j \end{cases}$ and make this simple,

$s_{ij} = \begin{cases} 1 \; if \; i = j \\ 0 \; if \; i \neq j \end{cases}$. So, the cases (i, j) = (1,1), (2,2), (3,3) calculates $s_i(1-s_j)$ and others calculates $-s_i \cdot s_j$.

d) Tanh function, derivative

The formula for the hyperbolic tangent function is tanh(z) = $\frac{e^z - e^{-z}}{e^z + e^{-z}}$ and tanh's range is between -1 and 1 compare to the sigmoid function's range between 0 and 1. At first, I wrote a code based on above formula, `np.exp(z)-np.exp(-z)/(np.exp(z)+np.exp(-z))`. But I got this,

```
([nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan],
 [18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18],
 [nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan],
 [32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32])
```

After this result, I fixed my code to `np.divide(np.exp(z)-np.exp(-z), np.exp(z)+np.exp(-z))`. I used np.divide() because it is elementwise divide to calculate the vector math. The result is:

```
…
1.5905691516276612,
  1.4643254632786735],
 [17, 17, 17, 17, 17, 17, 20, 17, 17, 17, 17, 17, 17, 17, 17],
…
  1.4493281226024253],
 [33, 33, 33, 33, 33, 33, 30, 33, 33, 33, 33, 33, 33, 33, 33])
```

And I wrote the derivative code based on this result:

$f'(x) = \frac{(e^x - e^{-x}) \cdot (e^x + e^{-x}) - (e^x - e^{-x}) \cdot (e^x + e^{-x})}{(e^x + e^{-x})^2} = 1 - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2} = 1 - f(x)^2$

e) ReLU function, derivative

We abbreviate Rectified Linear Unit to ReLU. From the NNDL book, the output of ReLU with input x, weight vector w, and bias b is $\max(0, w \cdot x + b)$. This function gradient still vanishes when z < 0 but mostly it works well. So, the code is `np.maximum(z, 0)`. To get the derivative, we need to think about two cases, z > 0 or z ≤ 0. When z is bigger than zero, z is going to be 1 and when z is smaller than or equal to zero, z is going to z. Based on this, I wrote, `np.where(z<=0, 0, 1)`.

f) def set_parameters

'set_parameter' function is setting hyperparameters and through this, we can change the cost function, activation function, regularization and dropoutpercent. I added code that changes the cost function to quadradic cost function if the output layer's activation function is tanh. Because the output value of tanh function is between 1 and -1. Also, I added code for the dropout. If we change the hyperparameter of dropoutpercent, the program notices from the flag and creates a list for the dropout layer.

g) def feedforward

Inside the feedforward function, I modified the code so that the code runs fine to work the hyperparameter of dropoutpercent setting other than the default values. I created the 'count'

variable that keeps tracking the number of dropout layers before the for loop. So, if we change the dropoutpercent, the code can reflect to hidden layers. Following the direction, the dropout applies to the hidden layers only. Also, for forward propagation, dropout should be applied to activation. Eventually, the network force to train with (dropoutpercent × the number of nodes in a hidden layer). These constraints help to reduce overfitting.

h) def SGD

This function trains the neural network using mini-batch stochastic gradient descent. The important part is calling update_mini_batch(). After training and before the testing/evaluation phrase, I added the code to set a flag to false, so it doesn't apply to the dropout during the testing/evaluation phrase. After calculating the evaluation cost and accuracy, I added the code to set the flag back to true if the dropoutpercent is not a default value.

i) def update_mini_batch

I modified the update_mini_batch() with two parts, one is for the dropout and the other is L1/L2 regularization. At first, for the dropout part, we already created the variable, 'd_outLs' to store nodes that were dropped in the set_parameters(). In this case, we change the dropoutpercent, if-condition will be the true and the code generates a dropout mask from this code: `u1 = np.random.binomial(1, self.dropoutpercent, size=(self.sizes[i],1)) / self.dropoutpercent` and then the dropout mask added the 'd_outLs' list. Also, it connects to the feedforward() and backprop(). The code, `a*=self.d_outLs[count-2]` is applying the mask to the activation of a hidden layer during the forward propagation.

The second part, regularization, of the code is hard-coding L2 regularization so, I separated L2 and L1 with if-else statement. L2 regularization is also called weight decay and Ridge. It ensures that the sum of squared weights is minimal. The sum of squares used here can be used to determine how much to constrain through the coefficient (lambda or alpha) in front of them (in the formula). Overall, we add an extra term to the cost function, and it helps to reduce overfitting. Because if the weights become large, it will scale down more and it keeps the individual values in weights from becoming too big. During gradient descent, the update step is: $w \to \left(1 - \frac{\eta\lambda}{n}\right) w - \frac{\eta}{m}\sum_x \frac{\partial C_x}{\partial w}$ and it exactly matched this code, `self.weights = [(1 - eta*(lmbda/n))*w -(eta/len(mini_batch))*nw for w, nw in zip(self.weights, nabla_w)`. $\left(1 - \frac{\eta\lambda}{n}\right)$ is a weight decay factor and it rescales the weight w. L1 regularization is also referred to as Least-Absolute Shrinkage and Selection Operator (LASSO). As the name suggests, the weights are reduced by constraining the sum of the absolute values of the weights to be minimum, and the weights converged to zero are created, which also functions as a selection. The update rule for L1 is:
$w \to w' = w - \frac{\eta\lambda}{n} sgn(w) - \eta \frac{\partial C_0}{\partial w}$. And I wrote the code, `self.weights = [w - (eta*(lmbda/n))*s - (eta/len(mini_batch))*nw for w, nw, s in zip(self.weights, nabla_w, signs)]`. We need to make 'sgn(w)' part work which is the sign of w. If w is positive, it is +1 and if w is negative, it is -1. I created a 'signs' list and it appends the result of positive and negative cases. So, whenever it updates w, it can remember the sign.

j) def backprop

In the feedforward part in the backprop(), I added code for the dropout like I did it in feedforward() because we forward-propagate the input x through the modified network and then backpropagate the result. We also need to go through the modified network. 'count' variable checks the dropout layer and if there is dropout, it applies the dropout mask to the activation of hidden layer.

k) def total_cost

To calculate the total_cost, we have two options of regularization, L1 and L2. I added an if-else condition and the code can calculate the cost by setting of L1 or L2 regularization. For example, this is the formular using cross-entropy cost function and L2 regularization.

$C = -\frac{1}{n}\sum_{xj}[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)] + \frac{\lambda}{2n}\sum_w w^2$

The code, `cost += self.cost.fn(a, y)/len(data)` gets the result from the cost function, it means this: $-\frac{1}{n}\sum_{xj}[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)]$. And then if we set the L2 regulation, we added above result and $\frac{\lambda}{2n}\sum_w w^2$. In the if-statement, `cost += 0.5*(lmbda/len(data)) * sum(np.linalg.norm(w)**2` for w in self.weights) calculates $\frac{\lambda}{2n}\sum_w w^2$. Else-statement calculates L1 regularization by adding the sum of the absolute values of the weights, $\frac{\lambda}{n}\sum_w |w|$.

3. Reflections

This assignment gave me a chance to modify the network.py code. I think it is somehow related to what I learned from the last assignment which is finding the best hyperparameter. In the beginning, I thought I had learned a lot from last assignment, and I was confident. I thought I knew enough about all the cost functions and how network code work. However, it took so much time to modify the code to get the results I was looking for. I got many errors and every time I changed the code; I need to recheck and execute the jupyter notebook to keep tracking my mistakes. For example, in the tanh function, I wrote code following the formula that I learned in class, but it didn't give me the right answer. I was so frustrated about why it didn't work because I was sure that the code was correct. I researched more and I realized that I didn't consider the elementwise operation. I learned the lesson that minor things are important and details matter. Finally, I received the results I expected. These results made me feel successful, complete, and excited. I learned more about numpy functions to write the cost functions and activation functions. I went through a lot of trial and error to make everything work. I wish I have more time to build graphs and plots to visualize all of the experiments. But this time, figuring out the network2.py code took most of my time, so I didn't have enough time to do that. This assignment was really challenging, and it also made me understand and learn more about the process of how the network calculates cost function and gets derivative values, feedforward, update and backpropagation clearly.