Sojeong Yang
CSC 578 Homework #4

Q1. Add comments for the following code snippet

```
# backward pass

delta = self.cost_derivative(activations[-1], y) * sigmoid_prime(zs[-1])
nabla_b[-1] = delta
nabla_w[-1] = np.dot(delta, activations[-2].transpose())
```

Line 1) `delta = self. cost_derivative(activations[-1], y) * sigmoid_prime(zs[-1])`
This line calculates the output error. This is equivalent to an equation from NNDL book, $\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$ and it is a componentwise expression, so we just use [*] to calculate. Also, it returns a shape (n,1). (n=number of nodes on the output layer). $\frac{\partial C}{\partial a_j^L}$ measures how quickly the cost is changing in jth output activation. In this case, jth output activation converted to code, activations[-1]. The activations[-1] means the last layer. 'y' is expected output (same as target output). To calculate it, we can do by this way, $\frac{\partial C}{\partial a_j^L} = (\partial a_j^L - y_j)$. The function inside the code, 'cost_derivative()' computes this part, $(\partial a_j^L - y_j)$. '`sigmoid_prime(zs[-1])`' is equivalent to an equation $\sigma'(z_j^L)$ from above and it calculates how quickly the activation function is changing at z_j^L. In this case, it uses zs[-1]. It means z vector of output layer.

Line2) `nabla_b[-1] = delta`
nabla_b means biases and biases are layer by layer list of numpy arrays and nabla_b[-1] is bias of the output layer. So, in this line of code, we are propagating the error to the bias. It is same as $\frac{\partial C}{\partial b_j^l} = \delta_j^l$ from NNDL book.

Line3) `nabla_w[-1] = np.dot(delta, activations[-2].transpose())`
It computes the rate of change of the cost with respect to weight in the network. And this line is equivalent to the equation, $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$ from NNDL book. 'nable_w'[-1] means weights of the output layer. 'activations[-2]' is that the activation of the neuron input to the weight and [-2] means the layer before the last layer. To calculate this, we need the matrix-based form to backpropagate. Suppose the network is […, B, A] that is the last layer output the vector of size B and the one before of size A. 'nable_w[-1]' will have [A, B] shape. 'delta' is the error of the neuron output from the weight and it will have [A, 1] shape. 'activations[-2] will have [B, 1] shape, therefore, we need to transpose it to [1, B]. The dot product makes that we want to do like this, [A, 1] * [1, B] → [A, B].

Q2. Exercise on Cross-entropy cost function in NNDL 3.
Show that the cross-entropy is still minimized when $\sigma(z) = y$ for all training inputs. When this is the case the cross-entropy has the value (Eq.64):

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

Sojeong Yang
CSC 578 Homework #4

- You can pick at least three values for y (between 0 and 1), and for each value of y, you should compute the Cross-entropy value using that y and several varying values of a (e.g. 0.1, 0.2, 0.3, …, 0.9).

  Choose three values of y: 0.2, 0.4, 0.7

y	a	$-y\ln(a)$	$-(1-y)\ln(1-a)$	$-y\ln(y)-(1-y)\ln(1-a)$
0.2	0.1	0.46051701859	0.08428841252	0.54480543111
	0.2	0.32188758248	0.17851484105	0.50040242353
	0.3	0.24079456086	0.28533995515	0.52613451601
	0.4	0.18325814637	0.40866049901	0.59191864538
	0.5	0.13862943611	0.55451774444	0.69314718055
	0.6	0.10216512475	0.73303258549	0.83519771024
	0.7	0.07133498878	0.96317824346	1.03451323224
	0.8	0.04462871026	1.28755032995	1.33217904021
	0.9	0.02107210313	1.8420680744	1.86314017753

y	a	$-y\ln(a)$	$-(1-y)\ln(1-y)$	$-y\ln(y)-(1-y)\ln(1-a)$
0.4	0.1	0.92103403719	0.06321630939	0.98425034658
	0.2	0.64377516497	0.13388613078	0.77766129575
	0.3	0.48158912173	0.21400496636	0.69559408809
	0.4	0.36651629275	0.30649537426	0.67301166701
	0.5	0.27725887222	0.41588830833	0.69314718055
	0.6	0.2043302495	0.54977443912	0.75410468862
	0.7	0.14266997757	0.72238368259	0.86505366016
	0.8	0.08925742052	0.96566274746	1.05492016798
	0.9	0.04214420626	1.3815510558	1.42369526206

y	a	$-y\ln(a)$	$-(1-y)\ln(1-y)$	$-y\ln(y)-(1-y)\ln(1-a)$
0.7	0.1	1.6118095651	0.03160815469	1.64341771979
	0.2	1.1266065387	0.06694306539	1.19354960409
	0.3	0.84278096302	0.10700248318	0.9497834462
	0.4	0.64140351231	0.15324768713	0.79465119944
	0.5	0.48520302639	0.20794415416	0.69314718055
	0.6	0.35757793663	0.27488721956	0.63246515619
	0.7	0.24967246075	0.36119184129	0.61086430204
	0.8	0.15620048592	0.48283137373	0.63903185965
	0.9	0.07375236096	0.69077552789	0.76452788885

According to the table, when y = 0.2, the minimum value is 0.50040242353 at a = 0.2. When y = 0.4, the minimum value is 0.67301166701 at a = 0.4 and when y = 0.7, the minimum value is 0.61086430204 at a = 0.7. All three cases support that the cross-entropy is minimized when $\sigma(z) = y$ following values of y.

Q3. Problem in NNDL Ch. 3: We (will have) discussed at length the learning slowdown that can occur when output neurons saturate, in networks using the quadratic cost to train. Another factor that may inhibit learning is the presence of the x_j term in Equation (61). Because of this term, when an input x_j is near to zero, the corresponding weight w_j will learn slowly. Explain why it is not possible to eliminate the x_j term through a clever choice of cost function.

Equation (61): $\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x x_j(\sigma(z) - y)$

In Equation (61), $\sigma(z) - y$ shows that the rate at which the weight learns is controlled by the error in the output. The output from the neuron is $a = \sigma(z)$, where $z = \sum_j w_j x_j + b$.
To work proper backpropagation, we calculate the cost as a function of the output activations. There are sigmoid, soft max, and linear activation functions used in neural networks. No matter which activation function is given, when the partial derivative of the cost function over the weight is taken, the derivative of the weight input always creates an additional x_j by the application of the chain rule. For example, cross-entropy cost function:

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

After compute the partial derivative of the cross-entropy cost with respect to the weights, $a = \sigma(z)$ so, we can get $\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_x (\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)})\frac{\partial \sigma}{\partial w_j} = -\frac{1}{n}\sum_x (\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)})\sigma'(z)x_j$.
The additional x_j cannot be removed due to the constraint that the cost should be a function of the output activation.

Q4. TensorFlow/Keras tutorial

<u>Must Not be changed:</u>

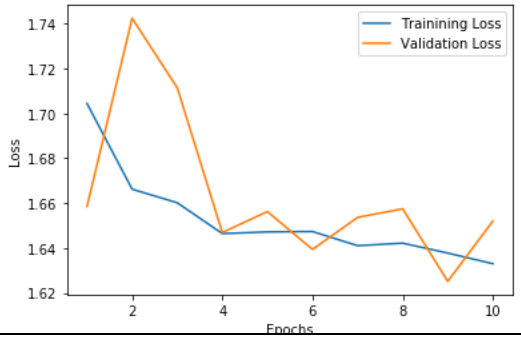
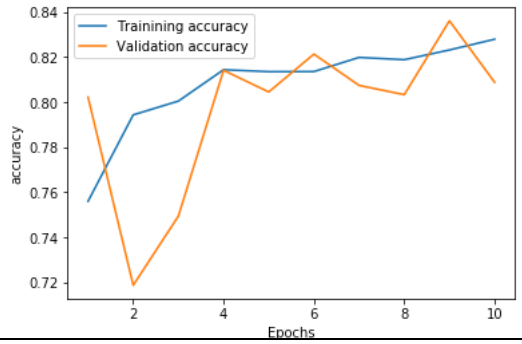
- The first layer has to be `Flatten` with `input_shape=(28, 28)` and the last layer has to be `Dense` with 10 nodes and `activation=tf.nn.softmax`
- In `compile()` the loss function has to be `sparse_categorical_crossentropy`

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
])
```

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

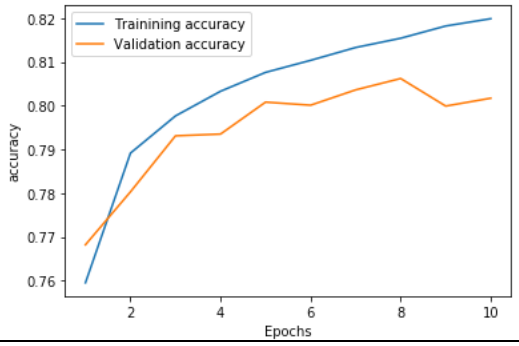
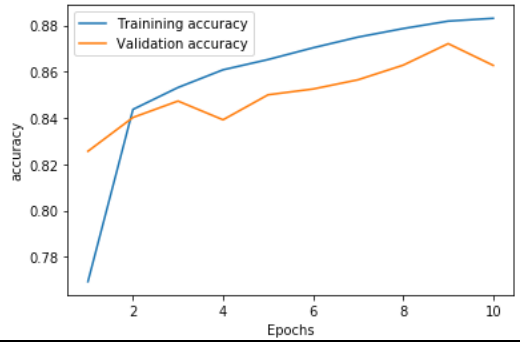
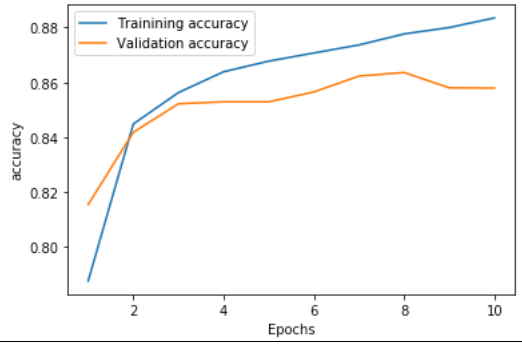
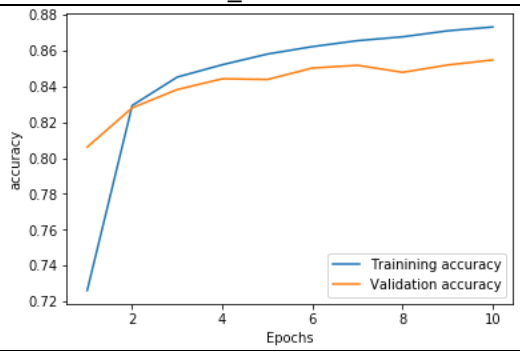
Sojeong Yang
CSC 578 Homework #4

I defined that 'best performance' is running our model with less overfitting since we learned about overfitting on the last assignment. So, I focused on searching for an optimal hyperparameter setting for minimizing overfitting and then checking the accuracy after a fixed number of epochs. First, I added 'validation_data' so, I can draw plots to keep track with the performance of the model.

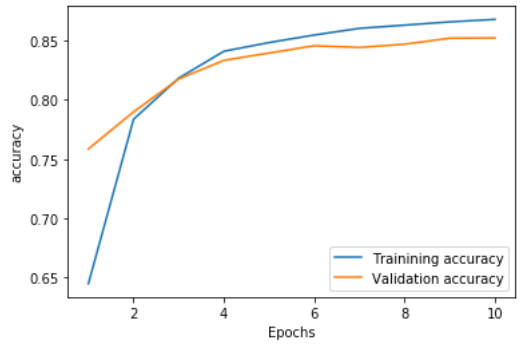
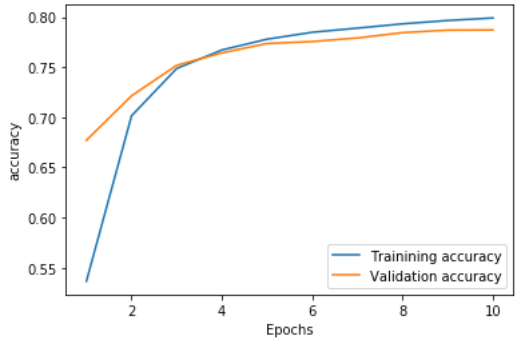
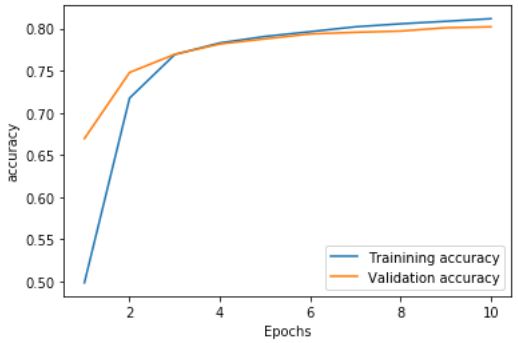
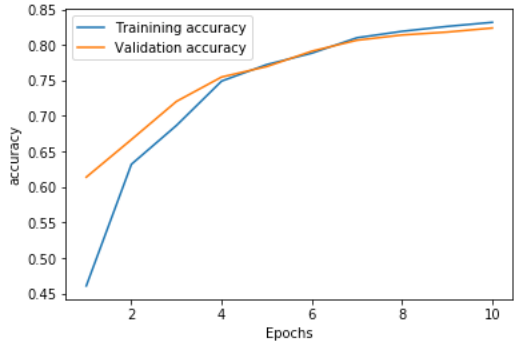
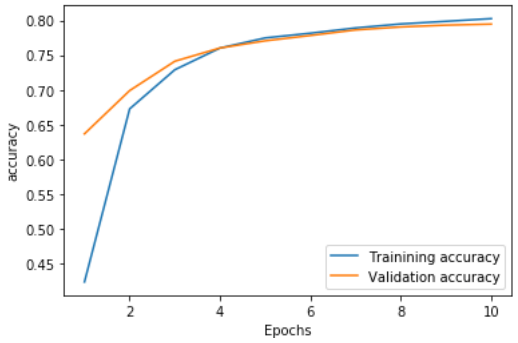
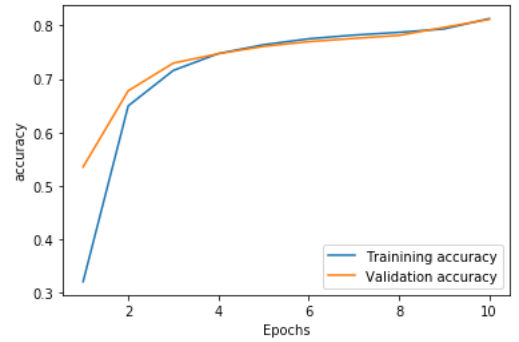
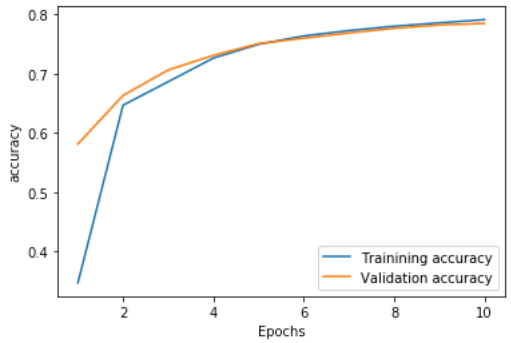
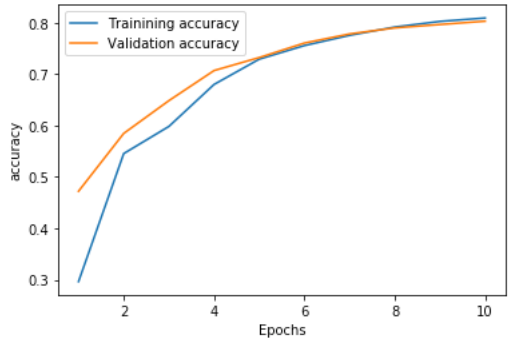
Given data with original setting	
Loss	Accuracy
	

We can see the overfitting the graph above with the original setting. The first parameter that I experimented with is 'mini-batch size' so, I kept changing a value 'batch-size' inside the model.fit(). I expected the good batch size is small so it can be updated more often.

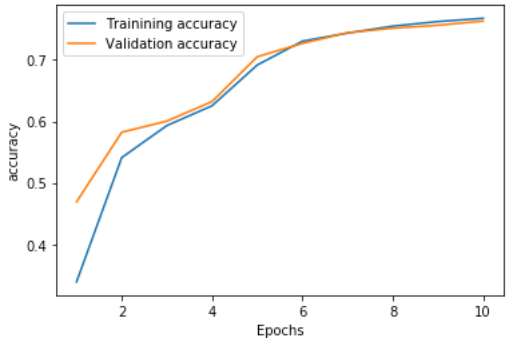
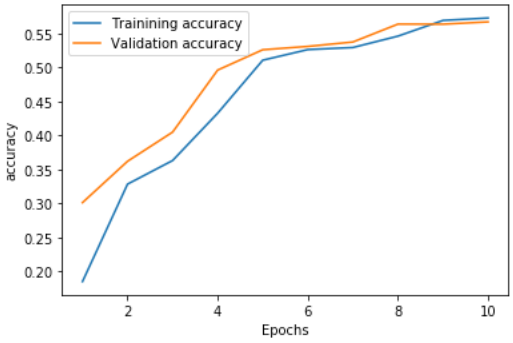
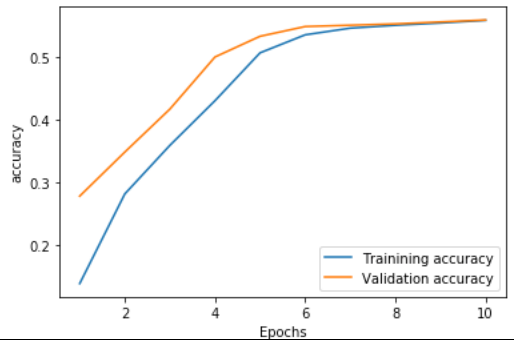
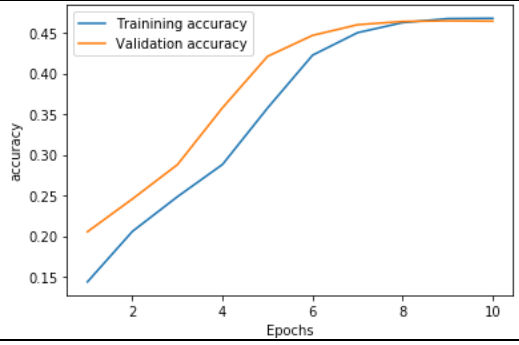
- Mini-batch size:

batch_size = 20	batch_size = 50
	
batch_size = 100	batch_size = 500
	

Sojeong Yang
CSC 578 Homework #4

batch_size = 1000	batch_size = 2000
	
batch_size = 3000	**batch_size = 4000**
	
batch_size = 5000	**batch_size = 6000**
	
batch_size = 7000	**batch_size = 8000**
	

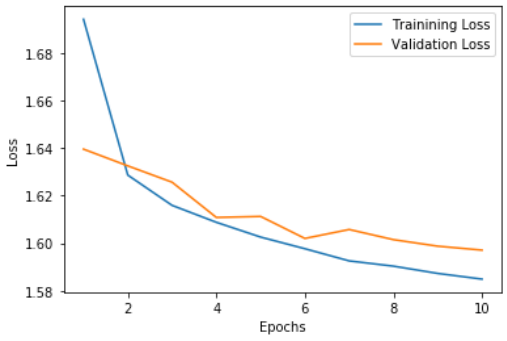
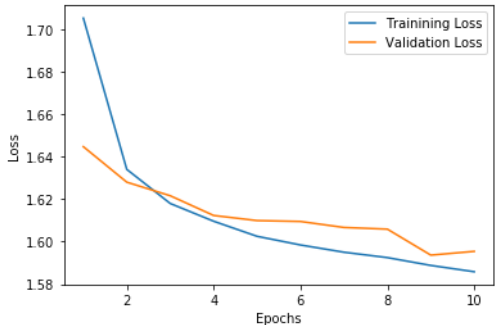
Sojeong Yang
CSC 578 Homework #4

batch_size = 10000	batch_size = 30000
	
batch_size = 40000	**batch_size = 60000**
	

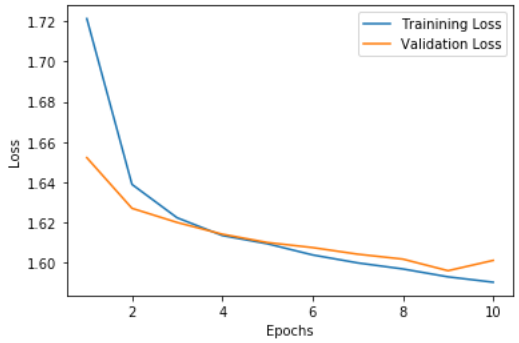
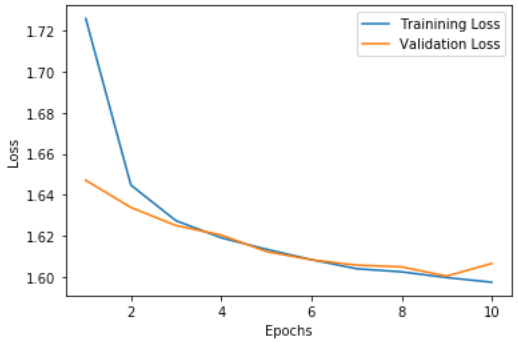
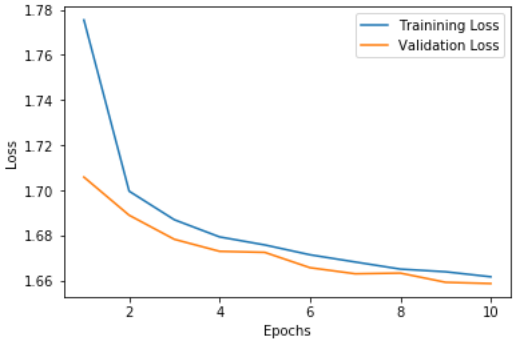
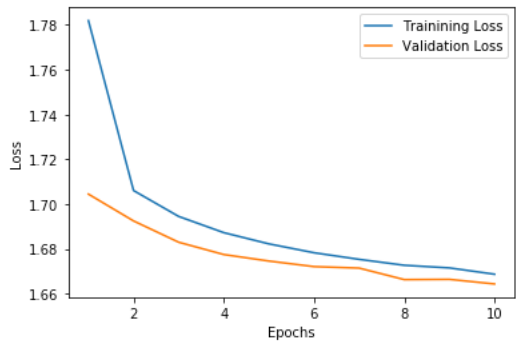
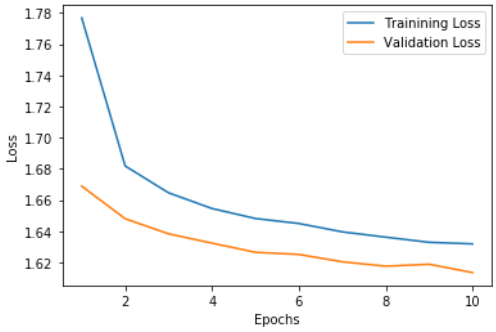
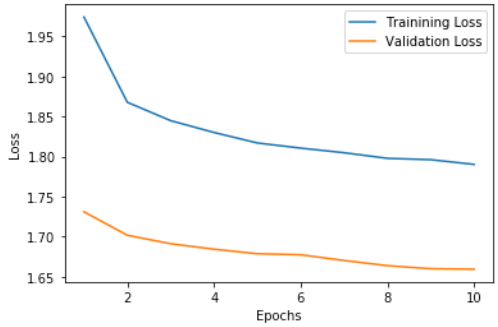
I can make the running speed up by using a larger batch-size and likewise, a small batch-size makes it run slower. It shows less overfitting between batch-size=3000 and 7000 but the accuracy is getting worse. From our NNDL textbook, the author wrote, “Choosing the best mini-batch size is compromise.”. If the mini-batch size is too small, we can’t get to take full advantage of the matrix library optimized for fast hardware. And if it’s too large, we cannot update the weights often. Therefore, we need to find a compromise for mini-batch size to speed up learning. Therefore, for less overfitting and high accuracy, I think I will experiment with a batch-size between 100 and 3000 with other hyper-parameters on the next step.

- Dropout:

I want to see the difference by changing the value of the dropout, so I ran a model by fixing the batch-size=100 first. I set a dropout value for the hidden layer. I expected that a dropout more than 50% is not efficient because it makes too robust features and it makes the model difficult to learn.

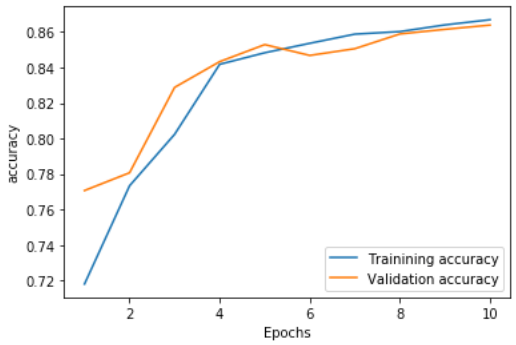
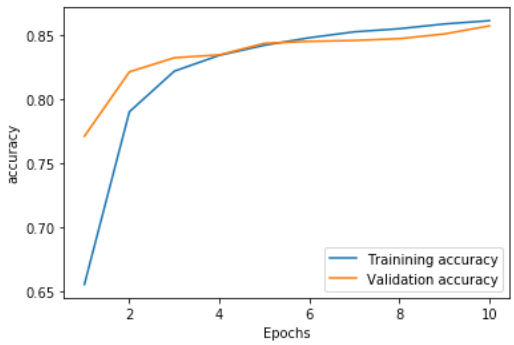
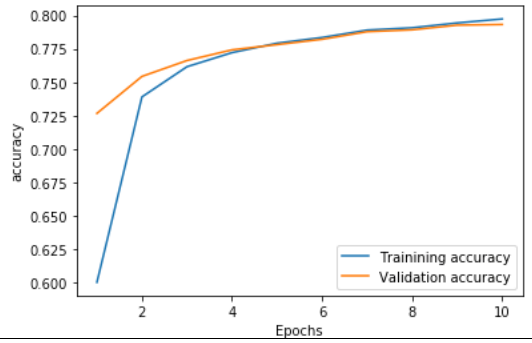
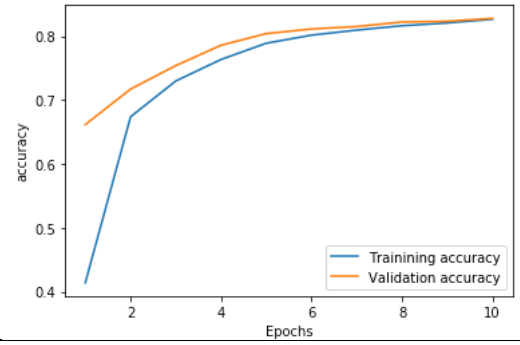
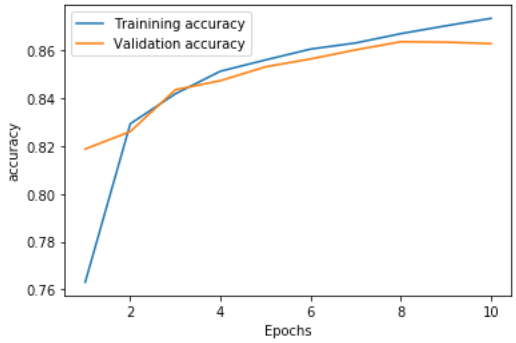
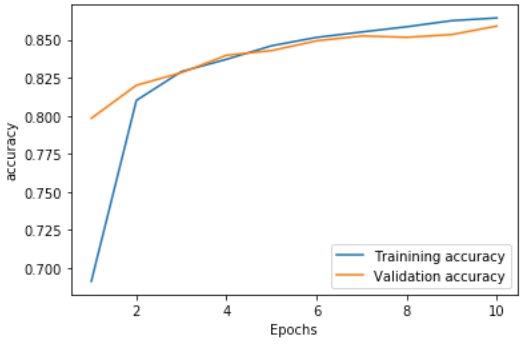
dropout (0.1)	dropout (0.2)
	

Sojeong Yang
CSC 578 Homework #4

dropout (0.3)	dropout (0.4)
	
dropout (0.5)	**dropout (0.6)**
	
dropout (0.7)	**dropout (0.9)**
	

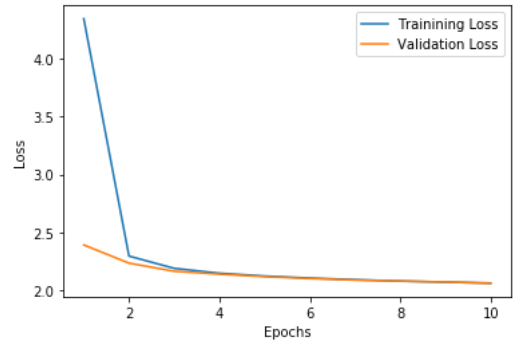
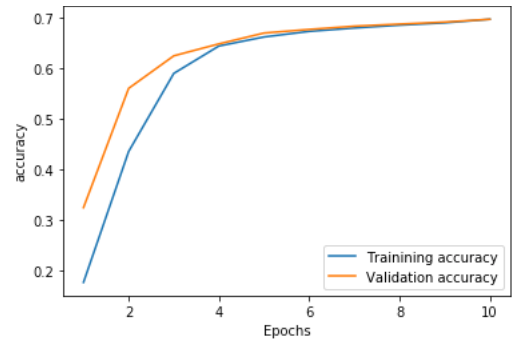
The result shows that dropout (0.4) has the minimum overfitting. Dropout reduces overfitting because the different networks can overfit in different ways, and averaging can help get rid of that kind of overfitting.

To search for an optimal model, I changed two values, batch-size and dropout depending on the result above. I tried a batch-size between 100 and 7000 and a dropout value of 0.4. It shows less overfitting, but the value of accuracy is not that high. So, I tried a different batch-size 100 and 500 with a dropout of 0.3. I found that the batch-size=100 and dropout 0.3 made pretty good accuracy and with less overfitting.

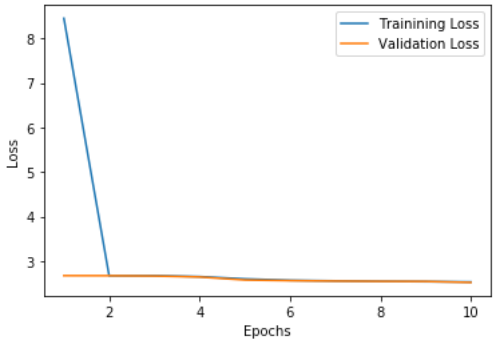
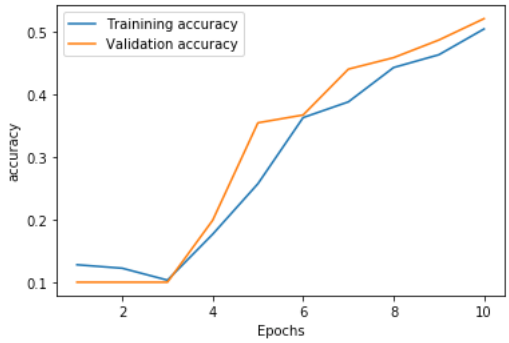
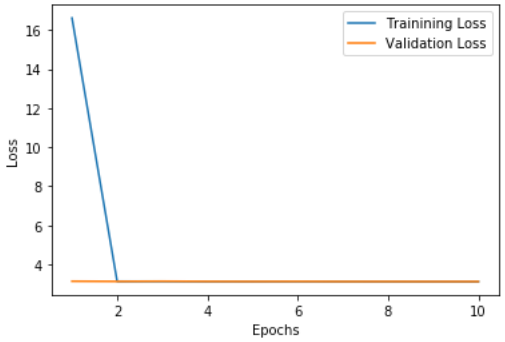
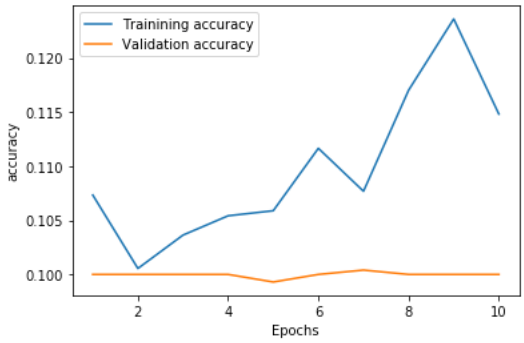
batch-size=100, dropout (0.4)	batch-size=500, dropout (0.4)
	
batch-size=1000, dropout (0.4)	**batch-size=3000, dropout (0.4)**
	
batch-size=100, dropout (0.3)	**batch-size=500, dropout (0.3)**
	

- Regularization methods: Using L1

These graphs are L1 regularization for hidden layer.

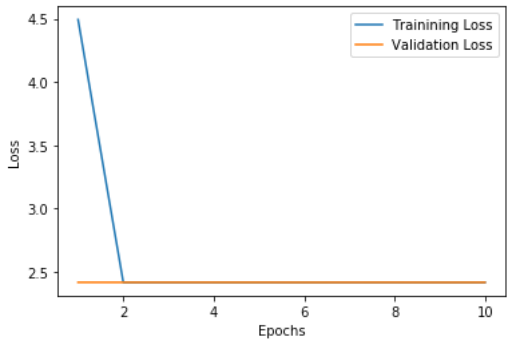
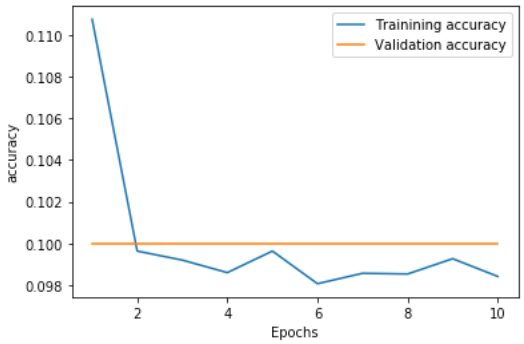
Loss, l=0.01(hidden layer)	Accuracy, l=0.01(hidden layer)
	

Sojeong Yang
CSC 578 Homework #4

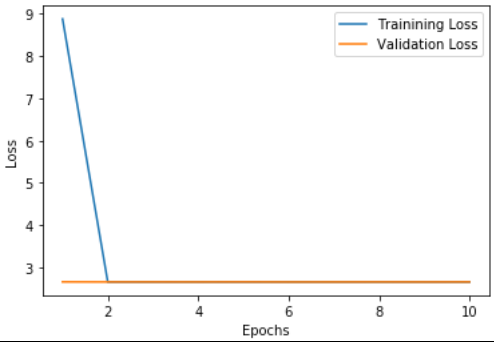
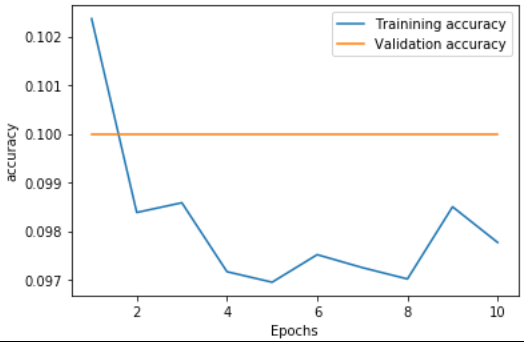
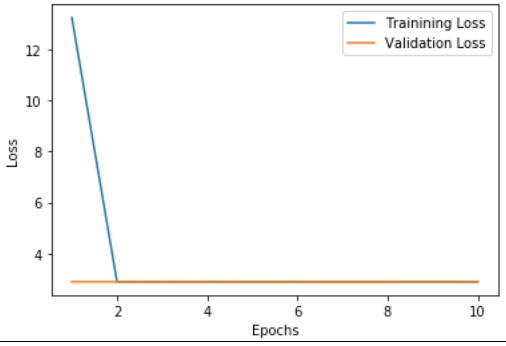
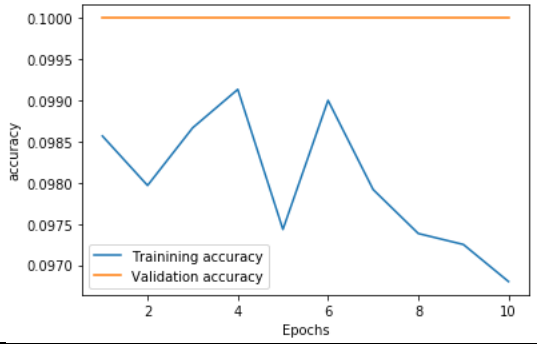
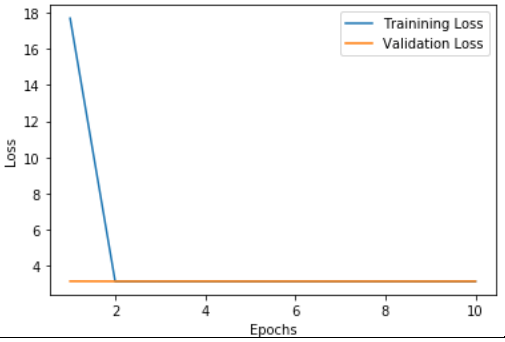
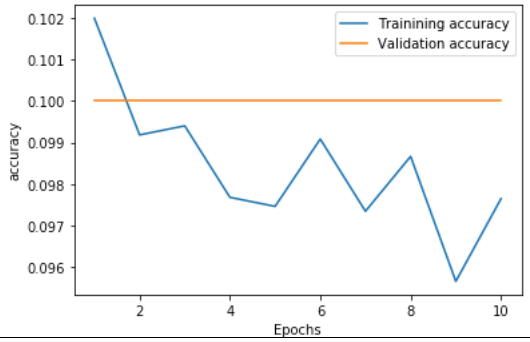
Loss, l=0.03(hidden layer)	Accuracy, l=0.03(hidden layer)
	
Loss, l=0.07(hidden layer)	**Accuracy, l=0.07(hidden layer)**
	

Here, I experimented with L1, L2 regularization with a batch-size=100. I wanted to know the results from using a L1 or L2 value on a different layer (hidden layer or output layer or both) and I tried it without setting a dropout value at first. The graphs show that the L1 regularization method reduces overfitting, but the value of accuracy varied depending on the value of L. It shows that the value of L for minimum overfitting is at L=0.01.

These graphs are L1 regularization for bot hidden layer and output layer.

Loss, l=0.01(hidden/output layer)	Accuracy, l=0.01(hidden/output layer)
	

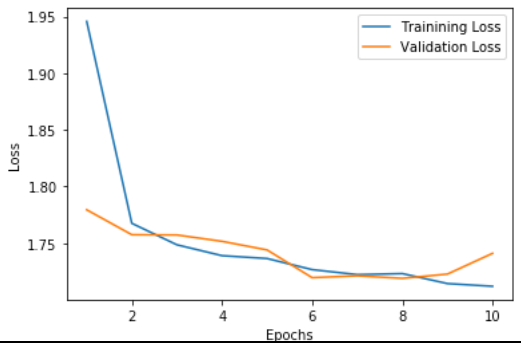
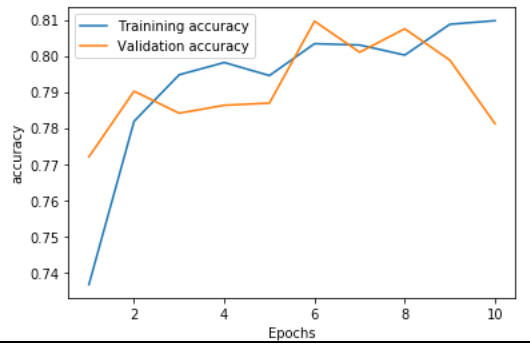
Sojeong Yang
CSC 578 Homework #4

Loss, l=0.03(hidden/output layer)	Accuracy, l=0.03(hidden/output layer)
	
Loss, l=0.05(hidden/output layer)	**Accuracy, l=0.05(hidden/output layer)**
	
Loss, l=0.07(hidden/output layer)	**Accuracy, l=0.07(hidden/output layer)**
	

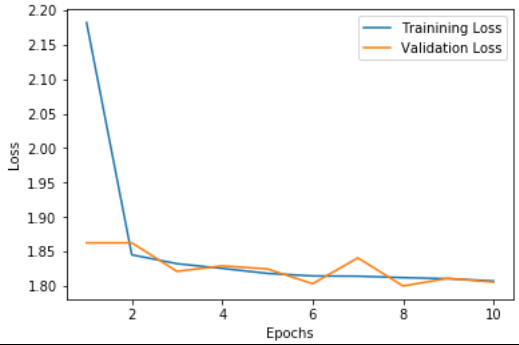
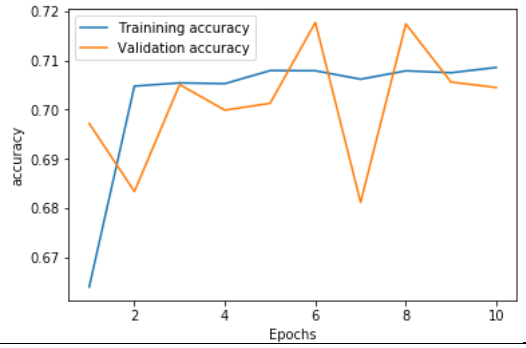
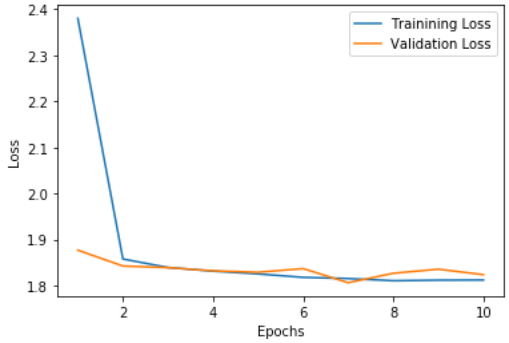
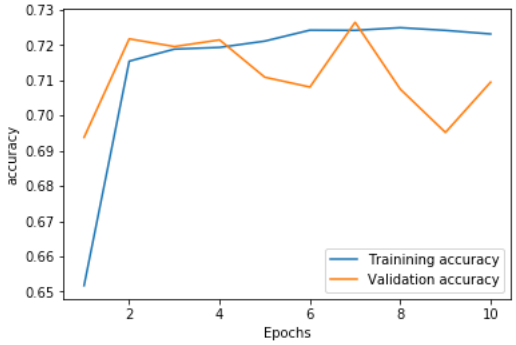
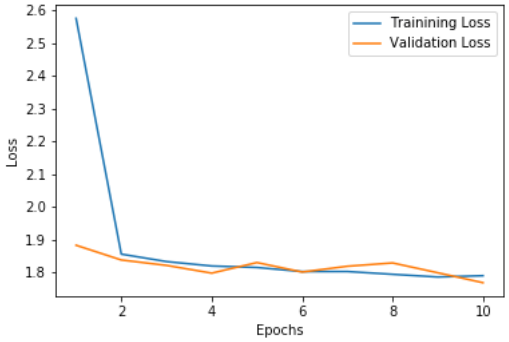
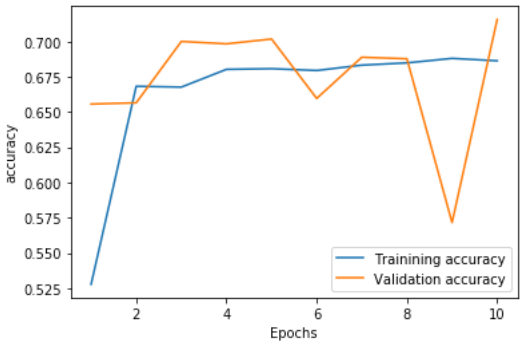
The loss value was similar, but accuracy was worse.

- Regularization methods: Using L2

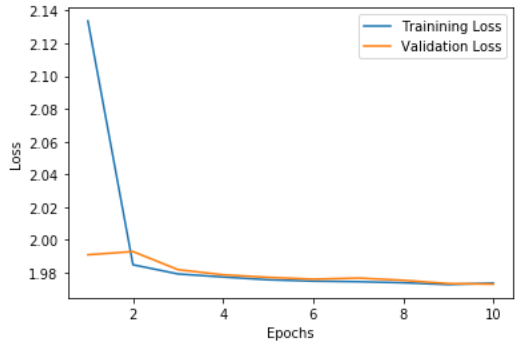
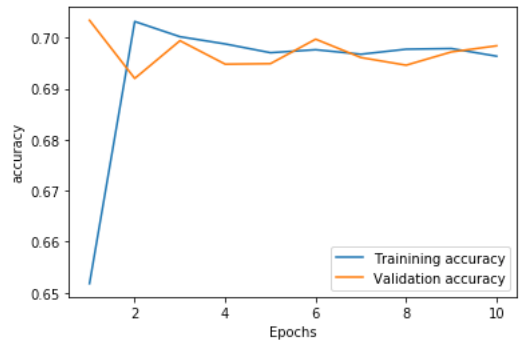
These graphs are L2 regularization for hidden layer.

Loss, l2=0.01(hidden layer)	Accuracy, l2=0.01(hidden layer)
	

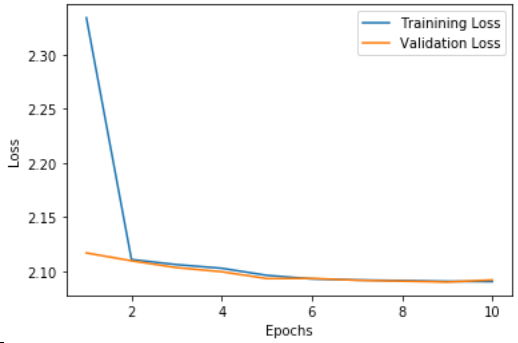
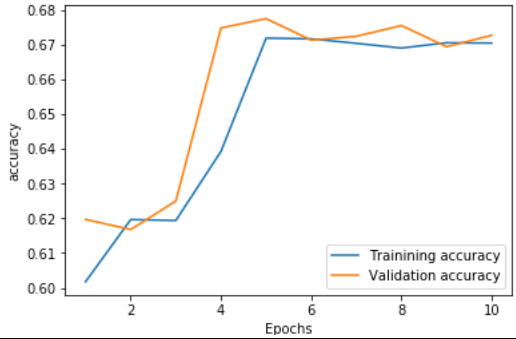
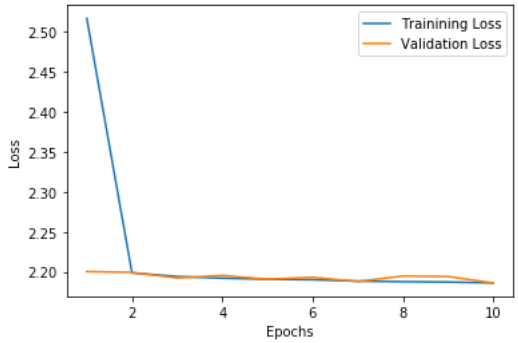
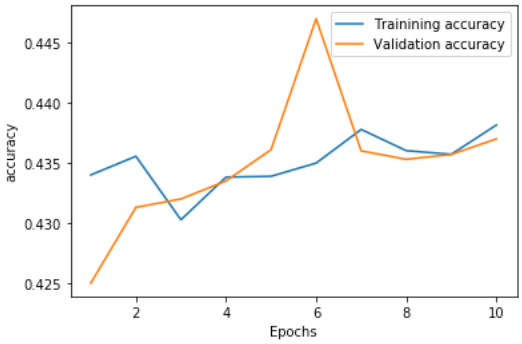
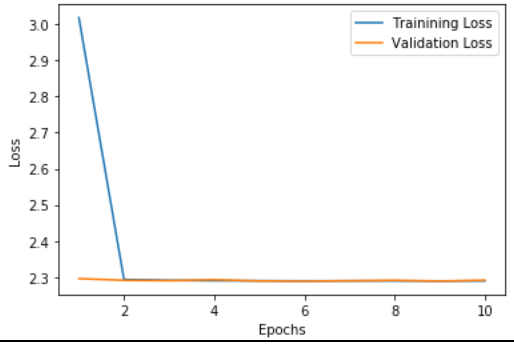
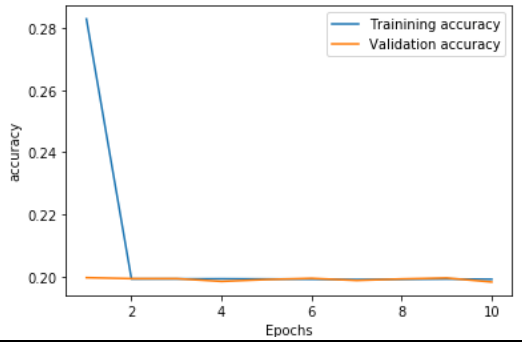
Sojeong Yang
CSC 578 Homework #4

Loss, l2=0.03(hidden layer)	Accuracy, l2=0.03(hidden layer)
	
Loss, l2=0.05(hidden layer)	**Accuracy, l2=0.05(hidden layer)**
	
Loss, l2=0.07(hidden layer)	**Accuracy, l2=0.07(hidden layer)**
	

These graphs are L2 regularization for both hidden layer and output layer.

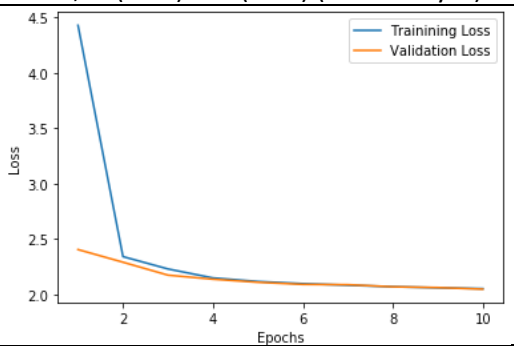
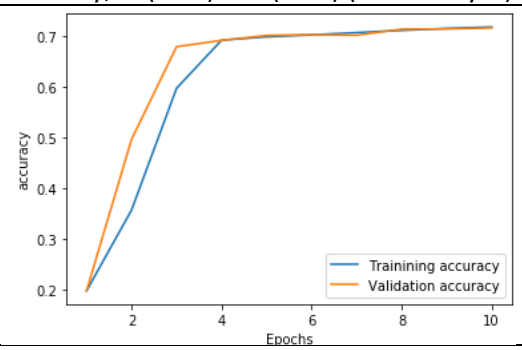
Loss, l2=0.01(hidden/output layer)	Accuracy, l2=0.01(hidden/output layer)
	

Sojeong Yang
CSC 578 Homework #4

Loss, l2=0.03(hidden/output layer)	Accuracy, l2=0.03(hidden/output layer)
	
Loss, l2=0.05(hidden/output layer)	**Accuracy, l2=0.05(hidden/output layer)**
	
Loss, l2=0.07(hidden/output layer)	**Accuracy, l2=0.07(hidden/output layer)**
	

Using L2 regularization to reduce overfitting and to get better accuracy than L1, the graphs shows that value of L2 at 0.01 for hidden/output layer is the best way.

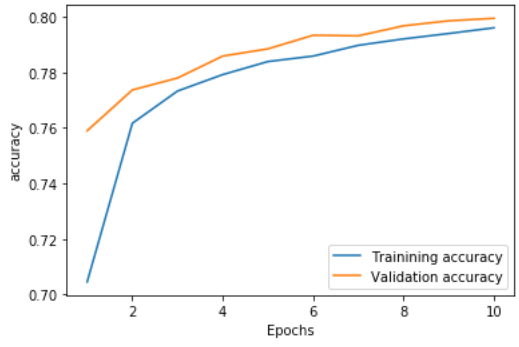
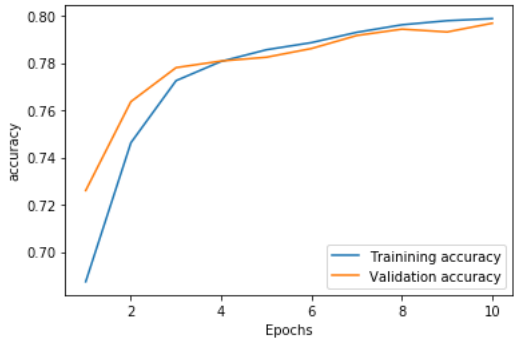
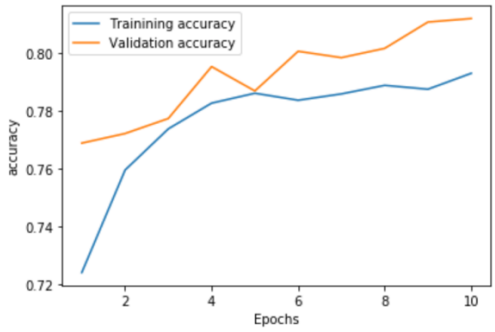
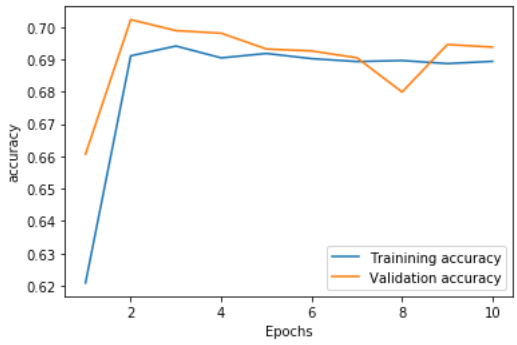
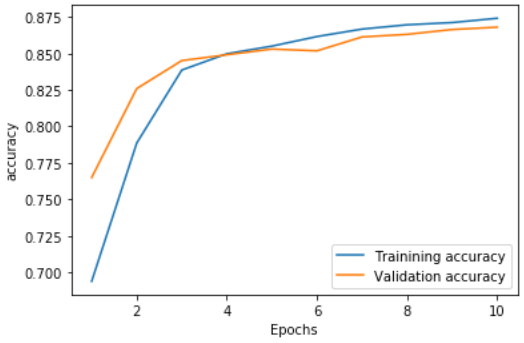
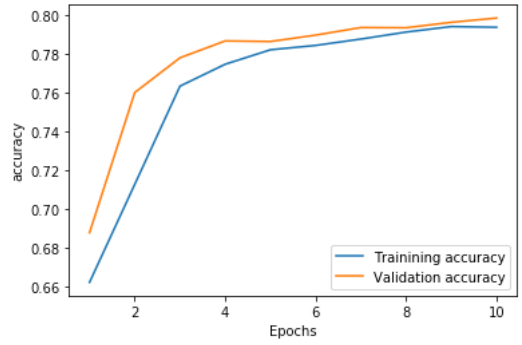
These graphs are L1+L2 regularization for hidden layer.

Loss, L1(0.01) + L2(0.01) (hidden layer)	Accuracy, L1(0.01) + L2(0.01) (hidden layer)
	

Sojeong Yang
CSC 578 Homework #4

I tried using a value 0.01 of L1 and L2 for hidden layer. They show less overfitting, but the accuracy is slightly less than using itself.

To search for an optimal model, I fixed two values, batch-size and dropout depending on the results above. And then, I tried L1=0.01 and L2=0.01 for output layer. Also, I tried L1+L2 for the output layer.

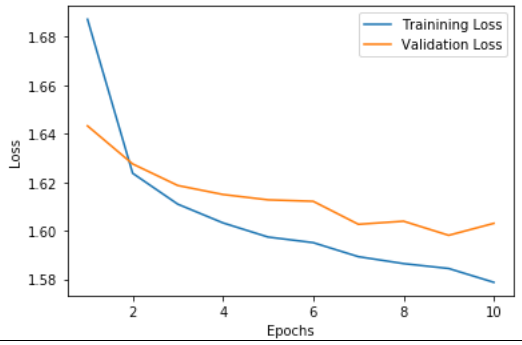
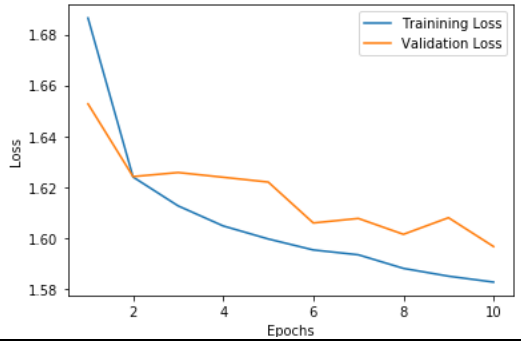
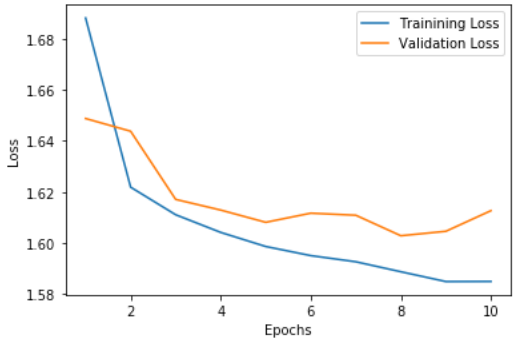
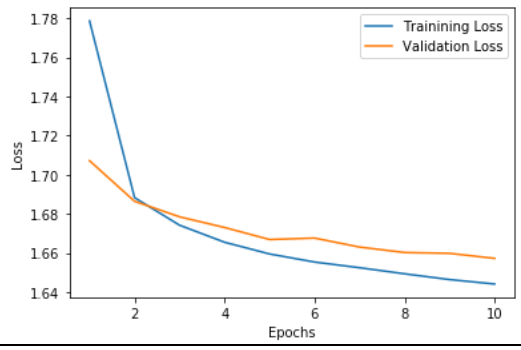
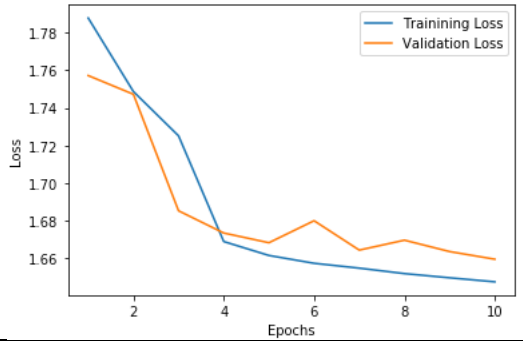
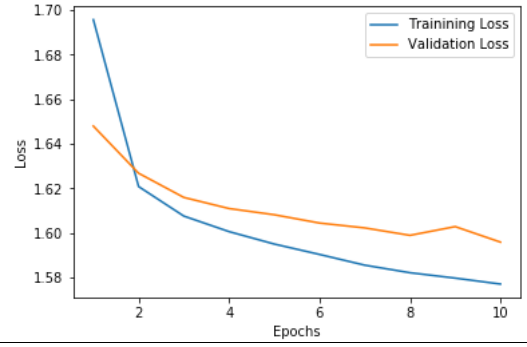
L1(0.01) (output layer)	L2(0.01) (output layer)
	
L2(0.01) (hidden layer)	**L2(0.01) (hidden/output layer)**
	
batch-size=100, drop (0.3)	**L1(0.01) + L2(0.01) (output layer)**
	

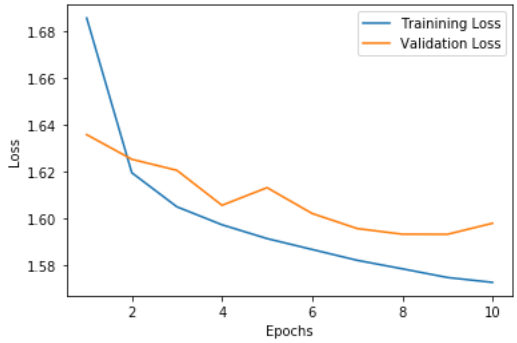
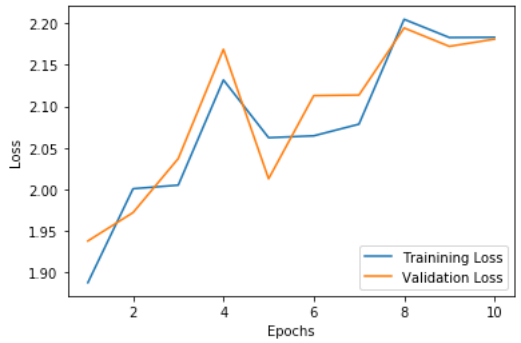
These graphs show that using L2(0.01) for the output layer and both L1(0.01) and L2(0.01) for the output layer get less overfitting and higher accuracy. But batch-size=100 and dropout 0.3 make it better accuracy without adding L1 and L2 regularization.

Next, I experimented with the activation function for hidden layers.

Sojeong Yang
CSC 578 Homework #4

- Activation function for hidden layers

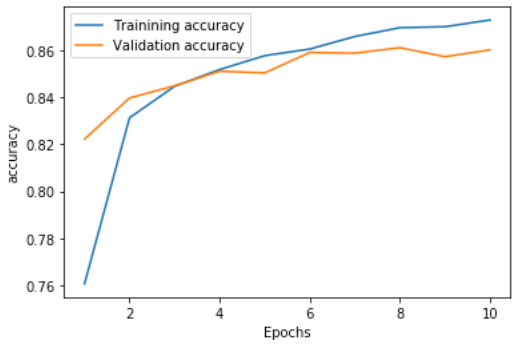
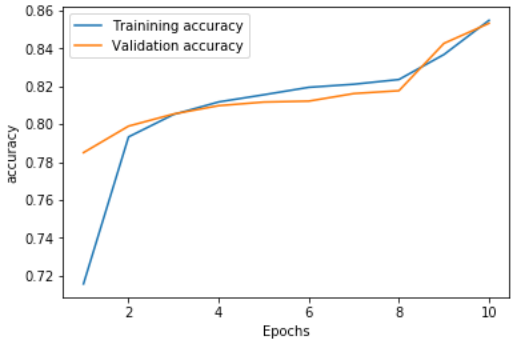
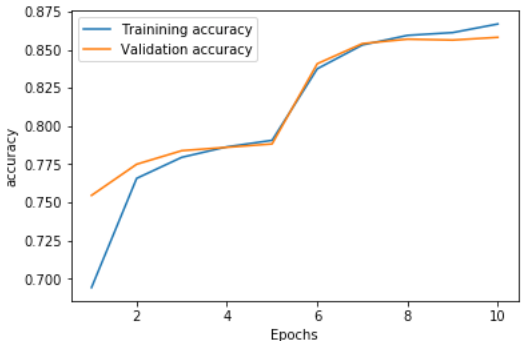
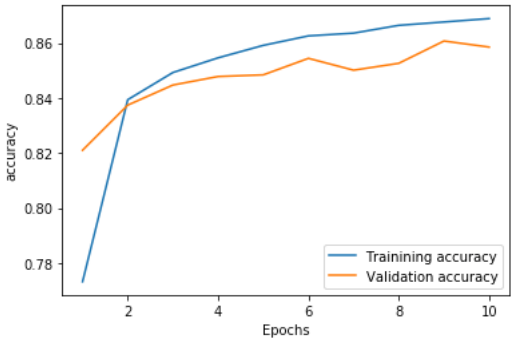
activation='relu'	activation='elu'
	
activation='selu'	**activation='sigmoid'**
	
activation='softplus'	**activation='softsign'**
	

activation='tanh'	activation='exponential'
	

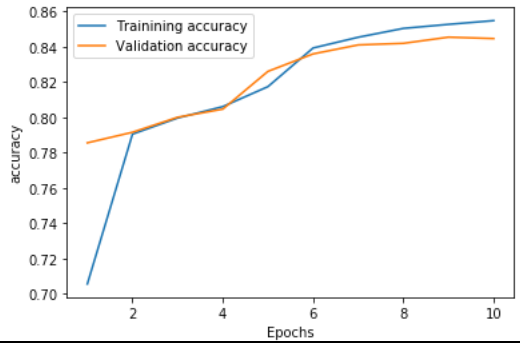
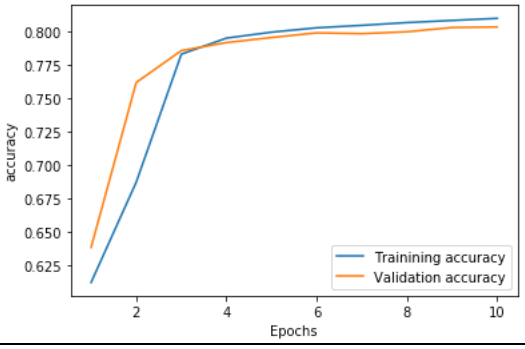
Sojeong Yang
CSC 578 Homework #4

After the experiment to set different activation function, ‘sigmoid’ is good to reduce overfitting with the parameter, batch-size=100.

To get the optimal model with this activation function, I experimented with other hyper-parameters that we got. (add dropout = 0.3)

activation='relu'	activation='sigmoid'
	
activation='softplus'	**activation='softsign'**
	

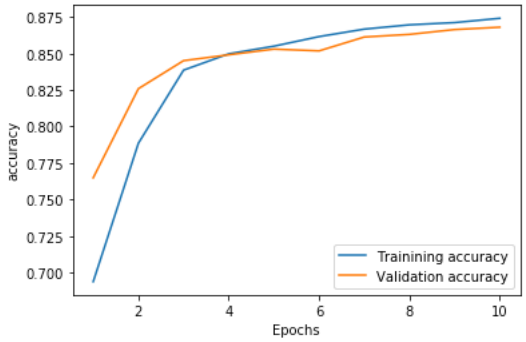
I experimented with batch-size=100, dropout = 0.3 and L2(0.01) for output layer.

activation='relu', L2(0.01)	activation='sigmoid', L2(0.01)
	

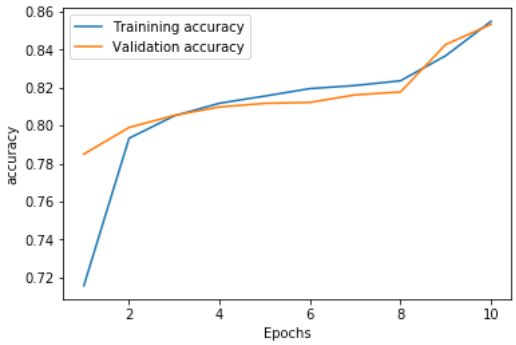
Sojeong Yang
CSC 578 Homework #4

Overall, the optimal model that satisfying less overfitting and high accuracy are:

- activation= ‘relu’, batch-size=100, dropout=0.3



- activation= ‘sigmoid’, batch-size=100, dropout=0.3



- activation= ‘relu’, batch-size=100, dropout=0.3, L2=0.01 for output layer