ECE421 – Introduction to Machine Learning

Assignment 2

Neural Networks

Hard Copy Due: Nov 6, 2019 in the class

Code Submission: Quercus webpage

Nov 6, 2019 @ 4:00 PM EST

General Notes:

- Attach this cover page to your hard copy submission
- Please post assignment related questions on Piazza.

Please check section to which you would like the assignment returned.

Tutorial Sections:

☐ Tutorial 1: Thursdays 1-3pm (GB304)
☐ Tutorial 2: Thursdays 4-6pm (BA1230)

Group Members	
Names	Student ID
Anas Akram Musharraf	1002428072
Anirudh Sampath	1002630218

1. Neural Networks Using Numpy

1.1 Helper Functions

These helper functions are required to implement a neural network. The helper functions we implement are:

1. The ReLU function. It returns the value of the input if the input is greater than or equal to zero, otherwise it returns zero. It is an activation function in neural networks.

```
def relu(x):
    return np.maximum(x,0)
```

2. The softmax function. The input is a matrix and it returns a matrix where each row is a probability distribution of the potential outcomes. In our case, each element in a row is the probability our input represents that specific letter.

```
def softmax(x):
    max_of_row = np.amax(x, axis=1).reshape(x.shape[0],1)
    a = np.exp(x-max_of_row)
    return a/np.sum(a, axis=1, keepdims=True)
```

3. The compute function. The inputs are the weight, input data, and bias matrices and it returns the matrix multiplication of the input data and the weight matrix, summed with the bias matrix.

```
def computeLayer(X, W, b):
    return np.matmul(X,W) + b
```

4. The average cross entropy function. The inputs are the labels and predictions. The functions returns the average cross entropy loss for the dataset.

```
def CE(target, prediction):
    interim = target * np.log(prediction)
    interim2 = np.sum(interim, axis=1)
    interim3 = np.sum(interim2, axis=0)
    return (-1/target.shape[0])*interim3
```

5. The gradient cross entropy function. The inputs are the labels and the input to the softmax functions. The functions returns the gradient of the cross entropy loss with respect to the input of the softmax function.

```
def gradCE(target, prediction):
    return softmax(prediction) - target
```

The derivation of the analytical solution[Appendix 1] can be simplified to:

$$\frac{\partial L}{\partial o} = p - y \tag{1}$$

Where p is the prediction matrix, y is the label matrix, and o is the input to the softmax equation.

1.2 Back Propagation Derivation

For the back propagation algorithm it is essential for us to know the analytical equations of the derivation of the cross entropy loss function with respect to the weights and biases of the hidden and outer layers [Appendix 2].

1. The gradient of the Loss with respect to the outer layer weights.

$$\frac{\partial L}{\partial W_o} = (p-y)(ReLU(W_h X + b_h)) \tag{2}$$

2. The gradient of the Loss with respect to the outer layer biases.

$$\frac{\partial L}{\partial b_o} = p - y \tag{3}$$

3. The gradient of the Loss with respect to the hidden layer weights.

$$\frac{\partial L}{\partial W_h} = (p-y)(W_o)(sign(ReLU(W_h X + b_h)))(X) \tag{4}$$

4. The gradient of the Loss with respect to the hidden layer biases.

$$\frac{\partial L}{\partial b_h} = (p-y)(W_o)(sign(ReLU(W_h X + b_h))) \tag{5}$$

1.3 Learning

We now trained the neural network using the number of epochs to be 200, a hidden unit size of 1000, gamma is 0.9, and learning rate is 1e-5. The weight matrices were first initialized using the Xavier initialization scheme.

Then using our helper functions, we were able to do a forward pass of the training data. Then using the gradients we find we do the back propagation algorithm to then update our weight and bias matrices. Lastly we then use our final weight and bias matrices on the training data and verify the accuracy by comparing our predictions with the actual labelled data.

Below are the plots of training, validating, and testing losses and accuracies over 200 epochs.

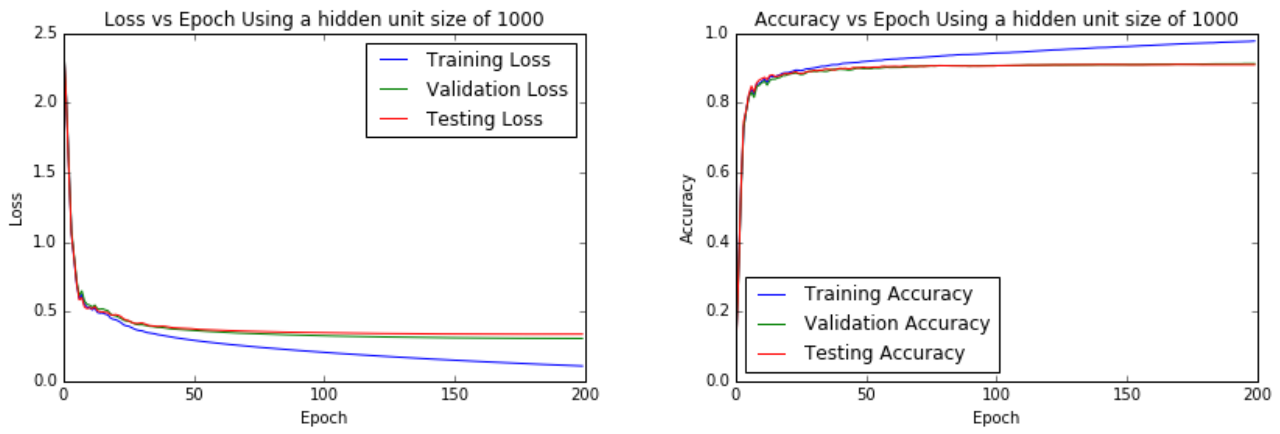


Figure 1: Training, validation and test losses and accuracies with 200 epochs, and a hidden layer size of 1000

Hidden Nodes	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Testing Loss	Testing Accuracy
1000	0.111	97.8%	0.332	91.3%	0.351	91.0%

Table 1: Training, validation and test losses and accuracies with 200 epochs, a hidden layer size of 1000, gamma of 0.9, and a learning rate of 1e-5

1.4 Hyperparameter Investigation

1.4.1 Number of Hidden Units

We investigate the effects of the size of the hidden layer by analyzing the changes in the training, validation, and test losses and accuracies when we change the size of the hidden layer. Below are the figures of the losses and accuracies for different sizes of the hidden layer.

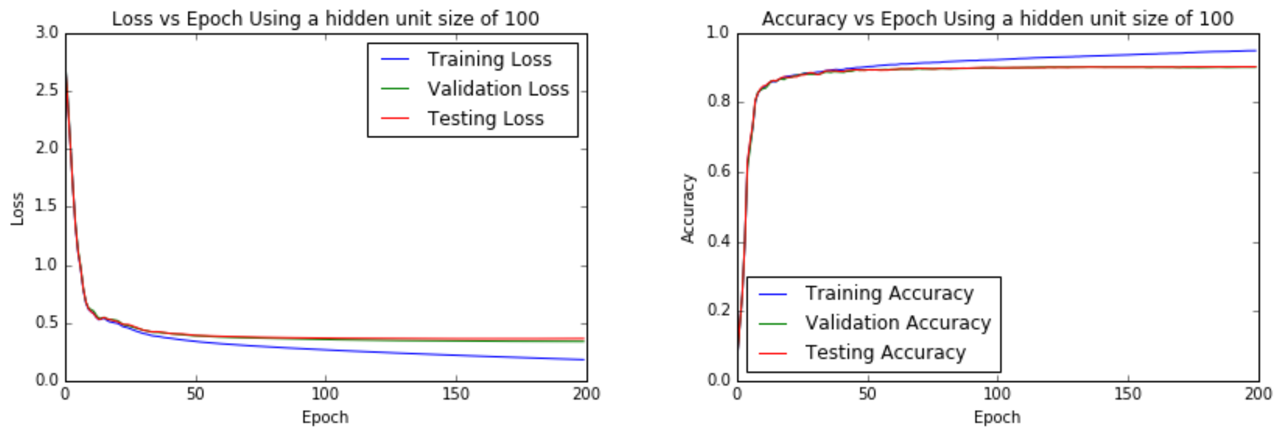


Figure 2: Training, validation and test losses and accuracies with 200 epochs, and a hidden layer size of 100

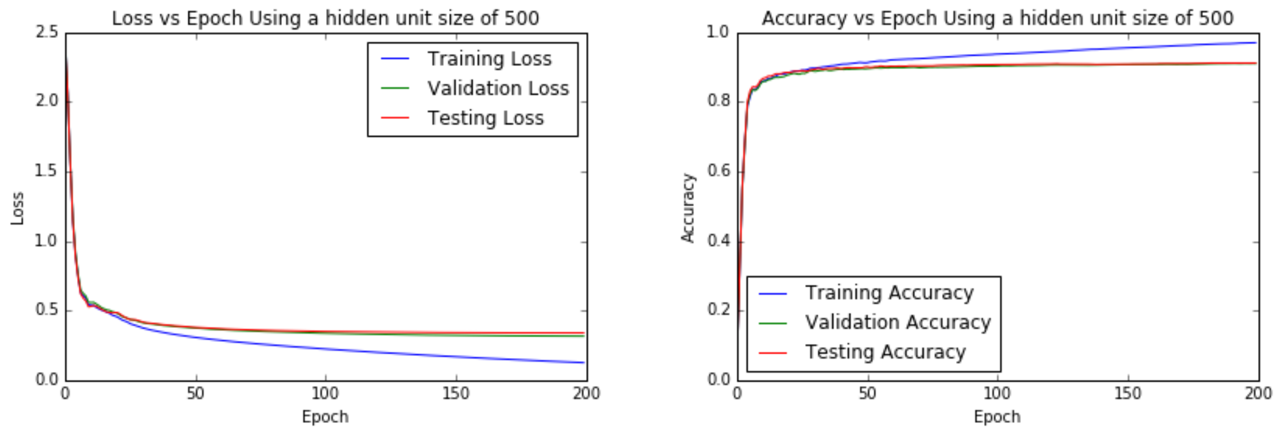
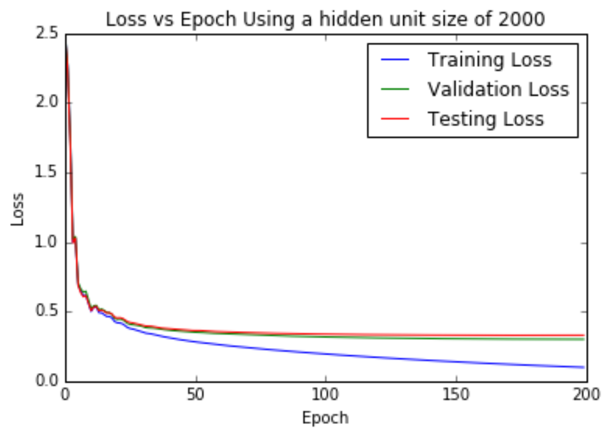


Figure 3: Training, validation and test losses and accuracies with 200 epochs, and a hidden layer size of 500



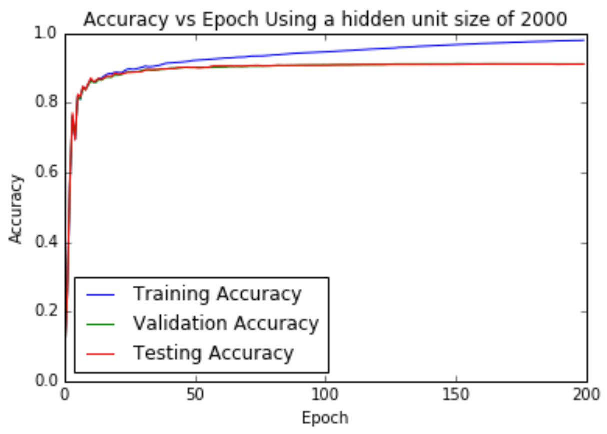


Figure 4: Training, validation and test losses and accuracies with 200 epochs, and a hidden layer size of 2000

To summarize all the information, below is a table is the values of the losses and accuracies with a different size of the hidden layer at the 200th epoch.

Hidden Nodes	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Testing Loss	Testing Accuracy
100	0.183	94.9%	0.330	90.2%	0.361	90.3%
500	0.127	97.0%	0.297	91.1%	0.357	90.7%
1000	0.111	97.8%	0.332	91.3%	0.351	91.0%
2000	0.101	98.0%	0.294	91.2%	0.337	90.9%

Table 2: Training, validation and test losses and accuracies with 200 epochs, a hidden layer size of 100, 500, 1000, and 2000.

Analyzing the graphs and the table of data it is clear that as the number of hidden layers increase, the training, validation, and test accuracies increase. However, this trend only lasts till 1000 hidden layer nodes, after that we see when we had 2000 hidden layer nodes, the validation and testing accuracy decrease.
Firstly we see the accuracies rise till 1000 because most probably below 1000 we were facing underfitting issues. That is when because of the fewer nodes, there are fewer weights which stops the model of accurately labelling the data.
On the contrary, we see after 1000 hidden layer nodes this trend does not persist. This is an indication of overfitting because we start having too many hidden layer nodes. This harms the models ability to generalize hence we see the accuracies only increase for training data. The model becomes very good at identifying the training data due to the excess number of nodes, but when faces with new data the model fails. Another issue with the large number of hidden layer nodes is that it takes a lot longer to run.

1.4.2 Early Stopping

The early stopping technique is to tackle the overfitting problem. It is when we stop iterating through the epoch once we see overfitting occurring. We know overfitting occurs once the trend of increasing accuracies as hidden nodes increase stop. In our case after analyzing the data, this seemed to happen at around 50 epochs. Below is a table summarizing the data at the 50th epoch.

Hidden Nodes	Training Accuracy	Validation Accuracy	Testing Accuracy
1000	92.4%	88.7%	86.9%

Table 3: Training, validation and test accuracies at 50 epochs, a hidden layer size of 1000.

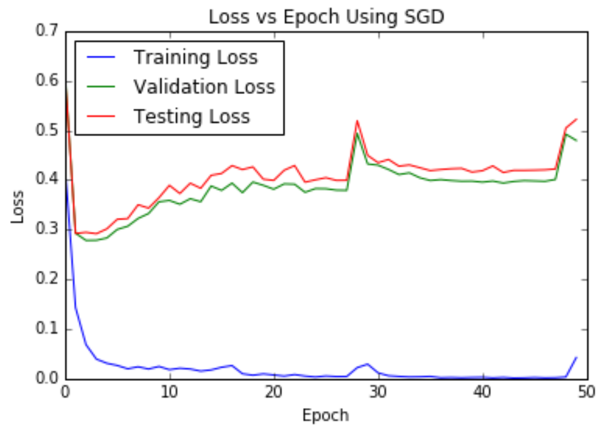
2. Neural Networks in Tensorflow

2.1 Model Implementation

```
def conv_neural_network():
    #data already comes out normalized
    trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()
    newTrain, newValid, newTest = convertOneHot(trainTarget, validTarget, testTarget)

    #reshaping all the matrices so they can be used in our model
    trainData = trainData.reshape(trainData.shape[0], 28, 28, 1)
    validData = validData.reshape(validData.shape[0], 28, 28, 1)
    testData = testData.reshape(testData.shape[0],28, 28, 1)

    #initializing a model variable, and state that we will add to it sequentially
    model = tf.keras.models.Sequential()

    #Creating a 3x3 convolutional layer with 32 filters, and using vertical and
    #horizontal strides of 1
    model.add(tf.keras.layers.Conv2D(32, kernel_size=3, activation=tf.nn.relu,
    strides=(1,1), input_shape=(28,28,1)))

    #Adding a batch normalization layer
    model.add(tf.keras.layers.BatchNormalization())

    #Adding a 2x2 max pooling layer
    model.add(tf.keras.layers.MaxPooling2D(pool_size = (2, 2)))

    #Adding a flattening layer
    model.add(tf.keras.layers.Flatten())

    #Creating a fully connected layer, and doing ReLU activation
    model.add(tf.keras.layers.Dense(784, activation=tf.nn.relu))

    #Creating a fully connected layer, and returning a softmax output
    model.add(tf.keras.layers.Dense(10, activation=tf.nn.softmax))

    #compiling the entire model, getting the cross entropy loss,
    #and using the ADAM optimizer
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
    loss='categorical_crossentropy',metrics=['accuracy'])
    model.fit(trainData, newTrain, batch_size=32, epochs=50)

    #Getting our train, valid, and test losses and accuracies
    test_loss, test_acc = model.evaluate(testData, newTest)
    val_loss, val_acc = model.evaluate(validData, newValid)
    train_loss, train_acc = model.evaluate(trainData, newTrain)

    #printing the results
    print(train_loss, train_acc)
    print(val_loss, val_acc)
    print(test_loss, test_acc)

    return 0
```

2.2 Model Training

Using our model, we used stochastic gradient descent for a batch size of 32, ran it for 50 epochs, and applied a learning rate of 1e-4 in the ADAM optimizer. The following are the graphs of our train, validation, and test losses and accuracies.



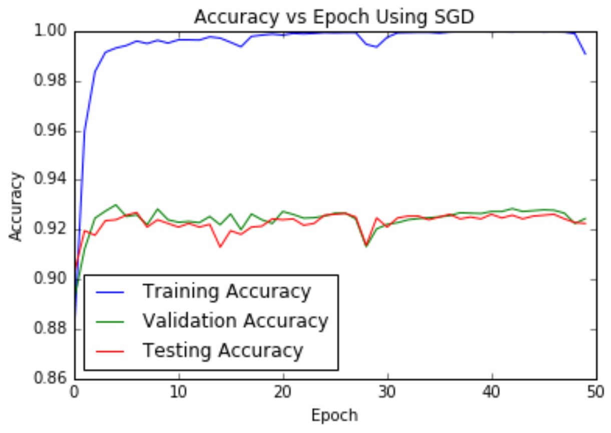


Figure 5: Training, validation and test losses and accuracies with Neural Network implemented using tensorflow

2.3 Hyperparameter Investigation

2.3.1 L2 Normalization

Weight Decay Coefficient	Training Accuracy	Validation Accuracy	Testing Accuracy
0.01	0.9887	0.9291667	0.914464
0.1	0.9547	0.924	0.9089574
0.5	0.9147	0.91183335	0.8682085

Table 4: Training, validation and test accuracies with different weight decay coefficients

The trend we notice is as the weight decay coefficient gets larger, the training, validation, and test accuracies get lower. Another trend is the validation, and test accuracies get closer to the training accuracy. Increasing the weight decay coefficient we prevent overfitting to occur, this is why the accuracies get closer to the training accuracy as we increase the weight decay coefficient. Furthermore, increasing the weight decay coefficient means we try to manage the magnitude of the weights instead of decreasing the error.

2.3.2 Dropout

The dropout method is another way to tackle the overfitting problem existing in neural networks with many hidden nodes. The dropout function takes a probability as an argument and this probability suggests how probable it is for a node to be ignored(dropped) or used to help train. We try different values for the probability, the table summarizing our findings and graphs are shown below.

Dropout Probability	Training Accuracy	Validation Accuracy	Testing Accuracy
0.9	0.9653	0.9355	0.8986784
0.75	0.9917	0.937	0.9155654
0.5	0.9979	0.93	0.924743

Table 5: Final training, validation and test accuracies with different dropout probabilities

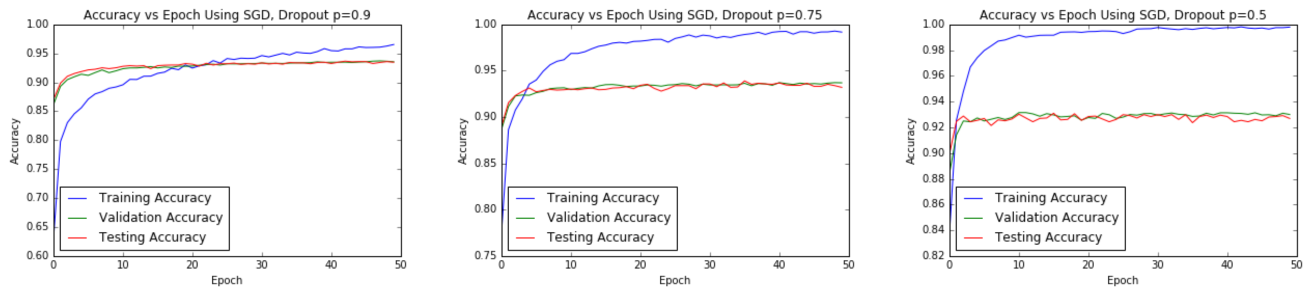


Figure 6: Training, validation and test accuracies of tensorflow neural network model, with different dropout probabilities

Analyzing the data, we can conclude that as the dropout probability decreases the model generalizes better. We get more consistent accuracies throughout the different data. This is expected because we are reducing the overfitting problem, we are reducing the number of nodes we consider hence we are generalizing the data better.

A

Dervation of gradCE

GradCE

$$\frac{\partial L}{\partial o} \qquad o = W_o h + b_o$$

$$h = \mathrm{ReLU}(W_h x + b_h)$$

Softmax: $\sigma_z = \dfrac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}} = p_j$

for $i=j$

$$\frac{\partial \sigma_i}{\partial z_j} = \frac{\partial \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}}}{\partial z_j} = \frac{e^{z_i}\sum_{k=1}^{N} e^{z_k} - e^{z_j}e^{z_i}}{\left(\sum_{k=1}^{N} e^{z_k}\right)^2} = \frac{e^{z_i}\left(\sum_{k=1}^{N} e^{z_k} - e^{z_j}\right)}{\left(\sum_{k=1}^{N} e^{z_k}\right)^2}$$

$$= \frac{e^{z_i}}{\sum_{k=1}^{N} e^{z_k}}\left[\frac{\sum_{k=1}^{N} e^{z_k}}{\sum_{k=1}^{N} e^{z_k}} - \frac{-e^{z_j}}{\sum_{k=1}^{N} e^{z_k}}\right]$$

$$= p_i(1 - p_j)$$

for $i \neq j$

$$\frac{\partial \sigma_i}{\partial z_j} = \frac{0 - e^{z_j}e^{z_i}}{\left(\sum_{k=1}^{N} e^{z_k}\right)^2} = -p_j p_i$$

So, $\dfrac{\partial p_i}{\partial z_j} = \begin{cases} p_i(1-p_j) & \text{if } i=j \\ -p_j p_i & \text{if } i \neq j \end{cases}$

$$L = -\sum y_i \log(p_i)$$

$$\frac{\partial L}{\partial \sigma} = -\sum_k y_k \frac{\partial \log(p_k)}{\partial \sigma_i} = -\sum_k y_k \frac{\partial \log(p_k)}{\partial p_k} \cdot \frac{\partial p_k}{\partial \sigma_i} = -\sum_k y_k \frac{1}{p_k} \cdot \frac{\partial p_k}{\partial \sigma_i}$$

So,

$$\frac{\partial L}{\partial \sigma} = -y_i(1-p_i) - \sum_{k\neq i} y_k \frac{1}{p_k}(-p_k p_i) = p_i\left(y_i + \sum_{k\neq i} y_k\right) - y_i$$

we know $\sum_k y_k = 1$ so, $y_i + \sum_{k\neq i} y_k = 1$

$$\therefore \boxed{\frac{\partial L}{\partial \sigma} = p_i - y_i}$$

B

Dervation of Backpropagation Algorithms

Backpropagation Derivations

$Z_h = W_h \cdot X + b_h \qquad a_h = \text{ReLU}(Z_h) \qquad Z_o = W_o \cdot a_h + b_o \qquad a_o = \text{softmax}(Z_o)$

$L = \sum_{k=1}^{k} y_k a_{o_k}$

$$\frac{\partial L}{\partial W_h} = \frac{\partial L}{\partial a_o} \cdot \frac{\partial a_o}{\partial Z_o} \cdot \frac{\partial Z_o}{\partial a_h} \cdot \frac{\partial a_h}{\partial Z_h} \cdot \frac{\partial Z_h}{\partial W_h} = (p-y)(W_o)(\text{sign}(\text{ReLU}(W_h X + b_h)))(X)$$

$$\frac{\partial L}{\partial b_h} = \frac{\partial L}{\partial a_o} \cdot \frac{\partial a_o}{\partial Z_o} \cdot \frac{\partial Z_o}{\partial a_h} \cdot \frac{\partial a_h}{\partial Z_h} \cdot \frac{\partial Z_h}{\partial b_h} = (p-y)(W_o)(\text{sign}(\text{ReLU}(W_h X + b_h)))$$

$$\frac{\partial L}{\partial W_o} = \frac{\partial L}{\partial a_o} \cdot \frac{\partial a_o}{\partial Z_o} \cdot \frac{\partial Z_o}{\partial W_o} = (p-y)(\text{ReLU}(W_h X + b_h))$$

$$\frac{\partial L}{\partial b_o} = \frac{\partial L}{\partial a_o} \cdot \frac{\partial a_o}{\partial Z_o} \cdot \frac{\partial Z_o}{\partial b_o} = p-y$$