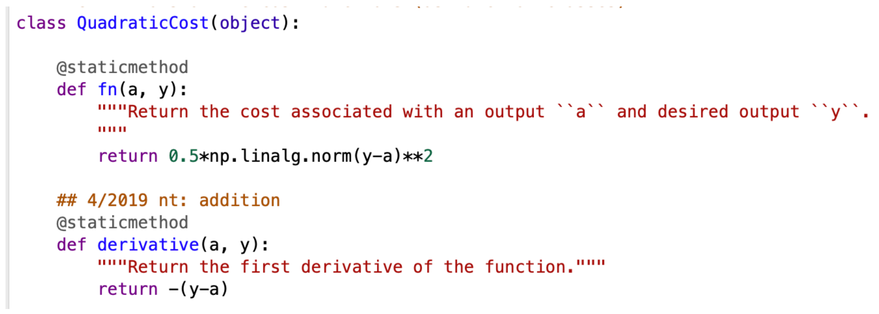
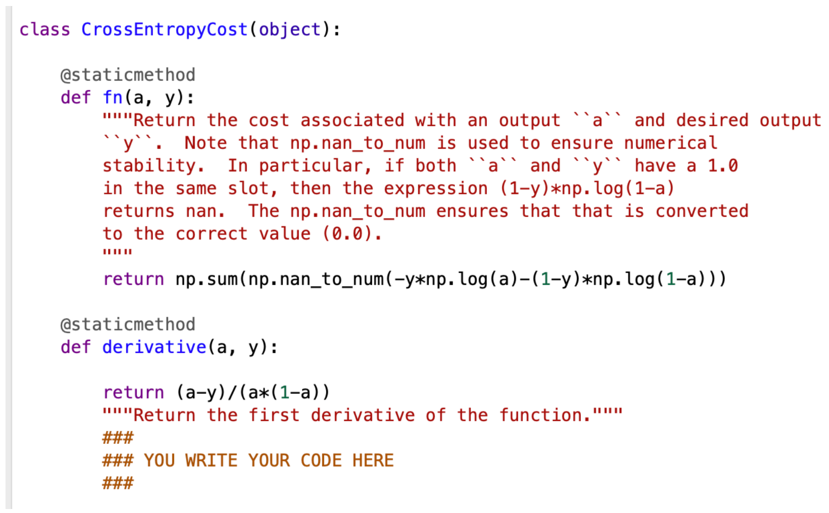
Kubra Iqbal
CSC 578
Explanation document

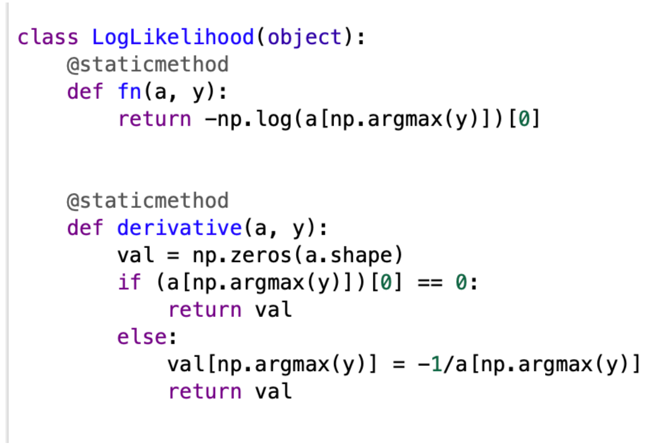
For the homework assignment – the goal was to match the first 10 experiments in the Application code which would give the results.
After adding code snippets and making changes to the Network File – I was able to have accurate results for all the 10 Experiments. (The experiments have been attached as an HTML file in the homework submission, so the answers can be compared). For experiments 11 and 12 – it is not exactly the same, but they are somewhat close to each other. Since it was mentioned that they are absolute minimal experiments – they did not match perfectly but gave approximate results.

To start with the Network code, there were changes to be made in hyper-parameters. The instructions state to start with Cost Function – which included Quadratic Cost, Cross Entropy and Loglikelihood.
The main reason to add a cost function is that it helps to measure how good a neural network was able to do with respect to its given training sample and expected output. It would also depend mostly on variables such as weights and bases.
The screen shots below show how I was able to implement this in the Network file that was provided.

Quadratic	$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$
Cross Entropy	$C = \frac{1}{n} \sum_x [-\mathrm{y(x)} \cdot \ln(a) - (1 - y(x)) \cdot \ln(1-a)]$
Likelihood	$C = \frac{1}{n} \sum_x -\ln(a_y^L)$ … usually used with Softmax activation function (for multiple output nodes)


```
class QuadraticCost(object):

    @staticmethod
    def fn(a, y):
        """Return the cost associated with an output ``a`` and desired output ``y``.
        """
        return 0.5*np.linalg.norm(y-a)**2

    ## 4/2019 nt: addition
    @staticmethod
    def derivative(a, y):
        """Return the first derivative of the function."""
        return -(y-a)
```


```
class CrossEntropyCost(object):

    @staticmethod
    def fn(a, y):
        """Return the cost associated with an output ``a`` and desired output
        ``y``.  Note that np.nan_to_num is used to ensure numerical
        stability.  In particular, if both ``a`` and ``y`` have a 1.0
        in the same slot, then the expression (1-y)*np.log(1-a)
        returns nan.  The np.nan_to_num ensures that that is converted
        to the correct value (0.0).
        """
        return np.sum(np.nan_to_num(-y*np.log(a)-(1-y)*np.log(1-a)))

    @staticmethod
    def derivative(a, y):

        return (a-y)/(a*(1-a))
        """Return the first derivative of the function."""
        ###
        ### YOU WRITE YOUR CODE HERE
        ###
```


```
class LogLikelihood(object):
    @staticmethod
    def fn(a, y):
        return -np.log(a[np.argmax(y)])[0]


    @staticmethod
    def derivative(a, y):
        val = np.zeros(a.shape)
        if (a[np.argmax(y)])[0] == 0:
            return val
        else:
            val[np.argmax(y)] = -1/a[np.argmax(y)]
            return val
```

The next step was to add and make changes in the act_hidden parameter. This parameter specifies the activation function for nodes on all hidden layers except the output layer.
The parameter options would include Sigmoid, Tanh, Relu and Softmax.
Adding these would help run the 10 experiments that were supposed to be done.

For the sigmoid – it takes a real value as input and outputs another value between 0 and 1. It is easier to work with and has all the properties of activation functions; it is non-linear, continuously differentiable, monotonic and has a fixed output range.

For the Tanh – the range for tanh is between -1 and 1 rather than 0 and 1. Tanh is also a transformation of sigmoid. Tanh's derivative if looked at mathametically is steeper than sigmoid. Which means oftentimes the learning is faster with tanh.

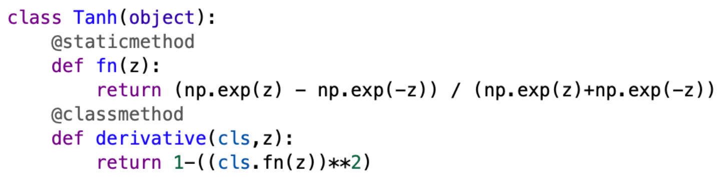
- <u>Definition</u>: $\tanh(z) = \dfrac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$

For the Relu – which is also known as the Ramp function. One of the advantages of this is that neurons do not saturate.

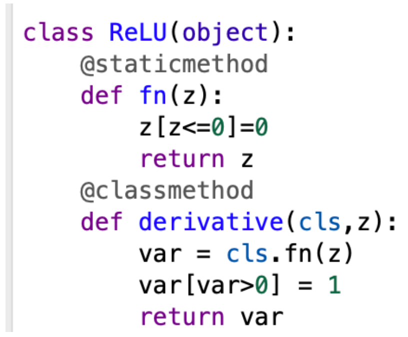
<u>Definition</u>: $f(z) = z^{+} = \max(0, z)$

For the Softmax – the function is usually applied to the output layer only. And this happens when there is more than one node on the output layer.

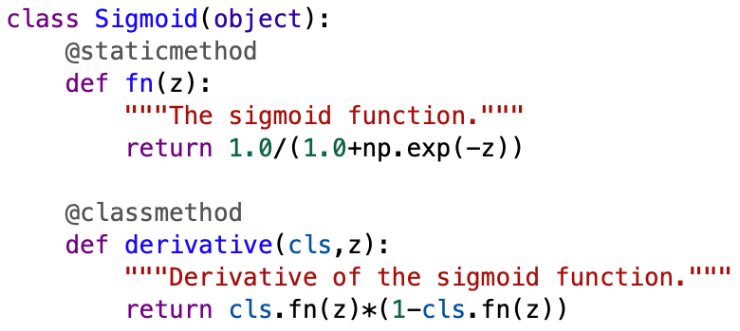
Below are the screenshots of what I added to these parameters.



```
class Tanh(object):
    @staticmethod
    def fn(z):
        return (np.exp(z) - np.exp(-z)) / (np.exp(z)+np.exp(-z))
    @classmethod
    def derivative(cls,z):
        return 1-((cls.fn(z))**2)
```



```
class ReLU(object):
    @staticmethod
    def fn(z):
        z[z<=0]=0
        return z
    @classmethod
    def derivative(cls,z):
        var = cls.fn(z)
        var[var>0] = 1
        return var
```



```
class Sigmoid(object):
    @staticmethod
    def fn(z):
        """The sigmoid function."""
        return 1.0/(1.0+np.exp(-z))

    @classmethod
    def derivative(cls,z):
        """Derivative of the sigmoid function."""
        return cls.fn(z)*(1-cls.fn(z))
```

```
class Softmax(object): #added this rn
    @staticmethod
    # Parameter z is an array of shape (len(z), 1).
    def fn(z):
        """The softmax of vector z."""
        #return (np.exp(z)/np.sum(np.exp(z)))
        '''Softmax activation function - Stable Version (Avoid ZERO numerator etc)'''
        return np.exp(z-np.max(z)) / np.sum(np.exp(z-np.max(z)))


    @classmethod
    def derivative(cls,z): #added this rn
        """Derivative of the softmax.
        IMPORTANT: The derivative is an N*N matrix.
        """
        a = cls.fn(z) # obtain the softmax activation vector
        return np.diagflat(a) - np.dot(a, a.T)
```

The next part of the problem was to work around with the Regularization which would specify the regularization method. Parameter options are L2 and L1 – this is particularly applied to all hidden layers and the output layer.

L1 – The term is added to the cost function so that over fitting can be avoided. This particular term has to be added not just to the cost in forward pass but also in the backpropagation, furthermore, the regularization value is also added to the cost function and its derivative is added to the weight update stage.
For L2, this was the default case already present in the code so no changes had to be made.

To learn and more about L2 Regularization, the aim to minimize the following cost function.

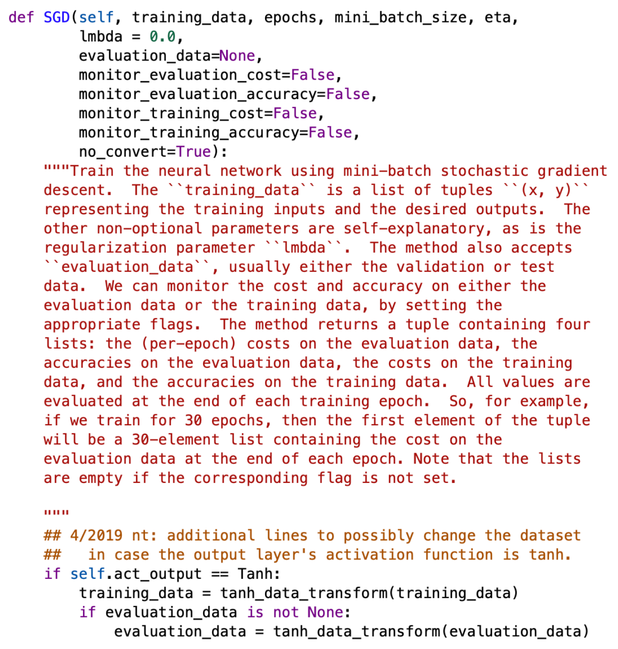
$$J\left(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}\right) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right)$$

Where L can be any loss function – we add a component that will penalize the weights.
Which would change the equation to be :

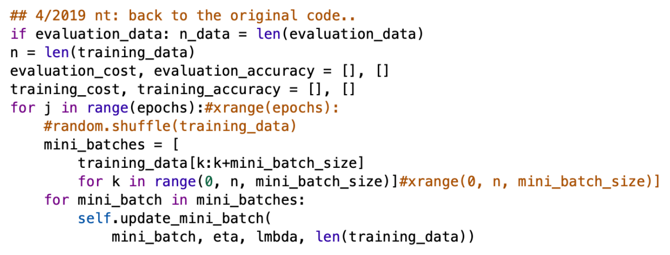
$$J\left(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}\right) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\sum_{l=1}^{L}\left\|w^{[L]}\right\|_F^2$$

The main reason of the regularization is – it helps in driving the values of the weight matrix down. By reducing the values in the weight matrix, Z will also be further reduced which would decrease the effect of the activation function. Therefore, a less complex function will be fit to the data and effectively reducing the overfitting of the data.

For Dropout Regularization – it involved going over all the layers in a NN and setting the probability of keeping a certain node or not. The input later and the output layer are kept the same in this scenario. The probability of keeping each node is set at a random, the value of the threshold is only determined if the node is kept or not. For example, if the threshold to 0.7 then there is a probability of 30% that a node will be removed from the given network.
On the other hand, the dropout means that the NN can not rely on any input node, since each have a random probability of being removed.
Therefore, the neural network will be reluctant to give high weights to certain features, because there is a chance that they might disappear. Consequently, the weights are spread across all features, making them smaller.



```
def SGD(self, training_data, epochs, mini_batch_size, eta,
        lmbda = 0.0,
        evaluation_data=None,
        monitor_evaluation_cost=False,
        monitor_evaluation_accuracy=False,
        monitor_training_cost=False,
        monitor_training_accuracy=False,
        no_convert=True):
    """Train the neural network using mini-batch stochastic gradient
    descent.  The ``training_data`` is a list of tuples ``(x, y)``
    representing the training inputs and the desired outputs.  The
    other non-optional parameters are self-explanatory, as is the
    regularization parameter ``lmbda``.  The method also accepts
    ``evaluation_data``, usually either the validation or test
    data.  We can monitor the cost and accuracy on either the
    evaluation data or the training data, by setting the
    appropriate flags.  The method returns a tuple containing four
    lists: the (per-epoch) costs on the evaluation data, the
    accuracies on the evaluation data, the costs on the training
    data, and the accuracies on the training data.  All values are
    evaluated at the end of each training epoch.  So, for example,
    if we train for 30 epochs, then the first element of the tuple
    will be a 30-element list containing the cost on the
    evaluation data at the end of each epoch. Note that the lists
    are empty if the corresponding flag is not set.

    """
    ## 4/2019 nt: additional lines to possibly change the dataset
    ##   in case the output layer's activation function is tanh.
    if self.act_output == Tanh:
        training_data = tanh_data_transform(training_data)
        if evaluation_data is not None:
            evaluation_data = tanh_data_transform(evaluation_data)
```



```
## 4/2019 nt: back to the original code..
if evaluation_data: n_data = len(evaluation_data)
n = len(training_data)
evaluation_cost, evaluation_accuracy = [], []
training_cost, training_accuracy = [], []
for j in range(epochs):#xrange(epochs):
    #random.shuffle(training_data)
    mini_batches = [
        training_data[k:k+mini_batch_size]
        for k in range(0, n, mini_batch_size)]#xrange(0, n, mini_batch_size)]
    for mini_batch in mini_batches:
        self.update_mini_batch(
            mini_batch, eta, lmbda, len(training_data))
            ## 4/2019 nt: from here, most lines are for printing purpose only.
            print ("Epoch %s training complete" % j)
            if monitor_training_cost:
                cost = self.total_cost(training_data, lmbda) # nt: for cost, always NO convert (default) for training
                training_cost.append(cost)
                print ("Cost on training data: {}".format(cost))
            if monitor_training_accuracy:
                accuracy = self.accuracy(training_data, convert=True) # nt: for accuracy, always _DO_ convert 
(argmax) for training
                training_accuracy.append(accuracy)
                print ("Accuracy on training data: {} / {}".format(
                    accuracy, n))
            if monitor_evaluation_cost:
                ## 9/2018 nt: changed the last parameter convert
                if no_convert:
                    cost = self.total_cost(evaluation_data, lmbda) # nt: if test/val data is already vectorized for y
                else:
                    cost = self.total_cost(evaluation_data, lmbda, convert=True)
                evaluation_cost.append(cost)
                print ("Cost on evaluation data: {}".format(cost))
            if monitor_evaluation_accuracy:
                ## 9/2018 nt: changed the last parameter convert
                if no_convert:
                    accuracy = self.accuracy(evaluation_data, convert=True) #nt: _DO_ convert (argmax)
                else:
                    accuracy = self.accuracy(evaluation_data)
                evaluation_accuracy.append(accuracy)
                print ("Accuracy on evaluation data: {} / {}".format(
                    ## 9/2018 nt: This seems like a bug!
                    #self.accuracy(evaluation_data), n_data))
                    accuracy, n_data))
            print ('')
        return evaluation_cost, evaluation_accuracy, \
            training_cost, training_accuracy #FIX IT -------------
```

For the dropout percent, this parameter would specify the percentage of the output. The value is between 0 and 1. Drop out function also consists in random setting and a fraction rate of units in a layer of 0 at each update during training time which would not cause over fitting.

In the training phase – with the Dropout, at each hidden layer, with the given probability we will kill the neuron. What it means by it is that the neuron will be set to 0. AS neural net is a collection of multiple operations, they won't propagate more.

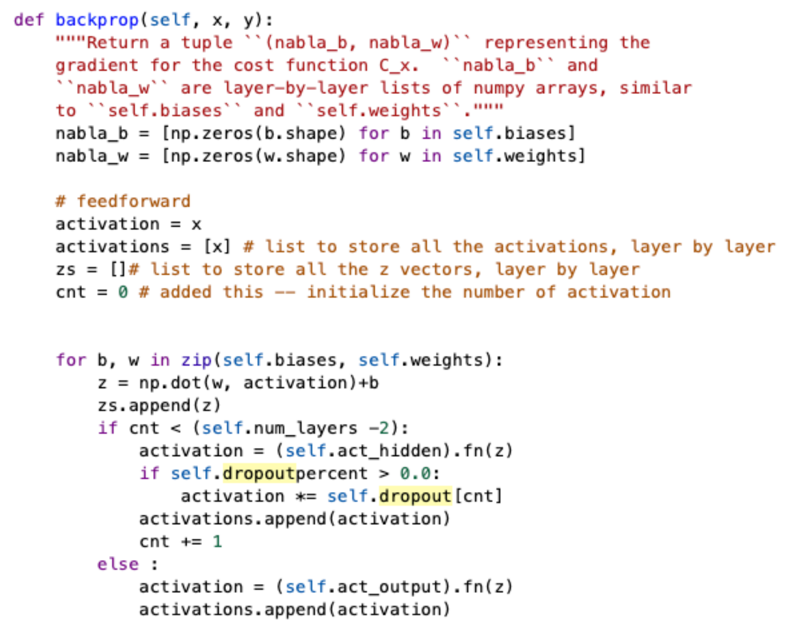
The drop out explains that there are hidden layers and we take out certain number of nodes which further depend on the dropout rate.
The point is that we keep relating this with different nodes and which node is bringing the network down, we should have just left the network like that.
We are maximizing the accuracy – drop out is another way of removing over fitting because we are not just making the nodes to do what we would have ideally liked them to do because we are applying the same rule.

For scenarios 11 and 12 – the accuracy increased gradually for training and testing.

```
    def __init__(self, sizes, cost=CrossEntropyCost, act_hidden=Sigmoid, \
                 act_output=None, regularization=None, dropoutpercent=0.0):
        """The list ``sizes`` contains the number of neurons in the respective
        layers of the network.  For example, if the list was [2, 3, 1]
        then it would be a three-layer network, with the first layer
        containing 2 neurons, the second layer 3 neurons, and the
        third layer 1 neuron.  The biases and weights for the network
        are initialized randomly, using
        ``self.default_weight_initializer`` (see docstring for that method).
        """
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.default_weight_initializer()
        self.cost=cost
        ## 4/2019 nt: addition
        self.act_hidden = act_hidden
        if act_output == None:
            self.act_output = self.act_hidden
        else:
            self.act_output = act_output
        self.regularization = regularization
        self.dropoutpercent = dropoutpercent
```

```
    def set_parameters(self, cost=QuadraticCost, act_hidden=Sigmoid, \
                       act_output=None, regularization=None, dropoutpercent=0.0):
        self.cost=cost
        self.act_hidden = act_hidden
        if act_output == None:
            self.act_output = self.act_hidden
        else:
            self.act_output = act_output

        # Overide act_output function to Sigmoid if act_output=Tanh and cost!=QuadraticCost
        if self.act_output == Tanh and self.cost != QuadraticCost:
            self.act_output = Sigmoid
            print("Activation function of Output Layer is overrided to Sigmoid function")

        #Overide Cost function to LogLikelihood when act_output function is Softmax
        if self.act_output==Softmax and self.cost != LogLikelihood:
            self.cost = LogLikelihood
            print("Cost Function is overridded to LogLikelihood because act_output is set as Softmax function")

        #Overide the activation function of hidden layer to Sigmoid function if hidden function is Softmax
        if self.act_hidden==Softmax:
            self.act_hidden=Sigmoid
            print("Activation function of Hidden Layer is overridded to Sigmoid Function")


        self.regularization = regularization
        self.dropoutpercent = dropoutpercent
        self.dropout = None # addition for the dropout mask
```

```
    def update_mini_batch(self, mini_batch, eta, lmbda, n):
        """Update the network's weights and biases by applying gradient
        descent using backpropagation to a single mini batch.  The
        ``mini_batch`` is a list of tuples ``(x, y)``, ``eta`` is the
        learning rate, ``lmbda`` is the regularization parameter, and
        ``n`` is the total size of the training data set.

        """
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]

        drop = [np.random.binomial(1,1-self.dropoutpercent, size=(self.sizes[i],1))/(1-self.dropoutpercent)
                 for i in range(1,len(self.sizes)-1)]
        self.dropout = drop

        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
```



```python
def backprop(self, x, y):
    """Return a tuple ``(nabla_b, nabla_w)`` representing the
    gradient for the cost function C_x.  ``nabla_b`` and
    ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
    to ``self.biases`` and ``self.weights``."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]

    # feedforward
    activation = x
    activations = [x] # list to store all the activations, layer by layer
    zs = []# list to store all the z vectors, layer by layer
    cnt = 0 # added this -- initialize the number of activation


    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        if cnt < (self.num_layers -2):
            activation = (self.act_hidden).fn(z)
            if self.dropoutpercent > 0.0:
                activation *= self.dropout[cnt]
            activations.append(activation)
            cnt += 1
        else :
            activation = (self.act_output).fn(z)
            activations.append(activation)
```

To summarize – My 10 experiments have matched accurately, when compared to the results provided by the professor. The HTML has everything saved, which should be easy to check.

Reflection:
I am really glad that we had two weeks to do this assignment. Even though it was pretty challenging, I did learn a lot from the concepts.
The most challenging part for me was the L1 – It took me longer to understand how to implement that part in the code. But once I understood what was being asked and how to take care of it, I was able to get that done.
I also struggled a little with backdrop, after dealing with the L1.
I was able to match the results from 1 to 10 – which was not that hard after I understood all the concepts and understood of how to make changes to the given network file.