Assignement_5

November 15, 2017

1 Training a Neural Network (1.0 points)

In this assignment, your task is to train a Neural Network or Multilayered Perceptron (MLP) classifier on synthetic data. Please read the assignment entirely before you start coding. Most of the code that implements the neural network is provided to you. What you are expected to do is fill in certain missing parts that are required and then train 2 different networks using gradient descent (Details below).

- Question 1 (0.2) Fill in the code to implement the sigmoid and relu functions and their derivatives. Next plot all four functions to certify your implementation is correct. See the example first
- Question 2 (0.8) Fill in the code inside the **Neural_Network** Class. Then train two networks one with sigmoid non-linearities for all the layers and one with relu non-linearities and softmax output.
- a. Complete the code in the feedforward method of the Neural_Network Class. Assuming that the input of a layer is a^{l-1} (which is the activation of a previous layer or the input feature vector) then the linear output of the layer is: $z^l = W * a^{l-1} + b$. Then the non-linear output or "activation" is $a^{l-1} = f(z^l)$ where $f()$ is the non-linearity of the layer. Use the gradient checking code (found here.) to ensure your additions to the code are working correctly. In our case this can be a sigmoid or a relu for hidden layers and it is always a sigmoid for the output layer.

- b. Explain briefly (a short paragraph) what is done by the for loop at the end of the **backprop** method of the **Neural_Network** Class.
- c. Train a network with sigmoid non-linearities across all layers. Then plot the accuracy and error curves. Finally visualize the estimated posterior. The code needed for you to do that is provided.
- d. By using the previous question's code as an example, train a network with relu non-linearities for the hidden layers and a sigmoid non-linearity for the output layer. Then plot the accuracy and error curves. Finally visualize the estimated posterior. The code for the last two tasks is provided. Then briefly compare on the performance of the two networks.

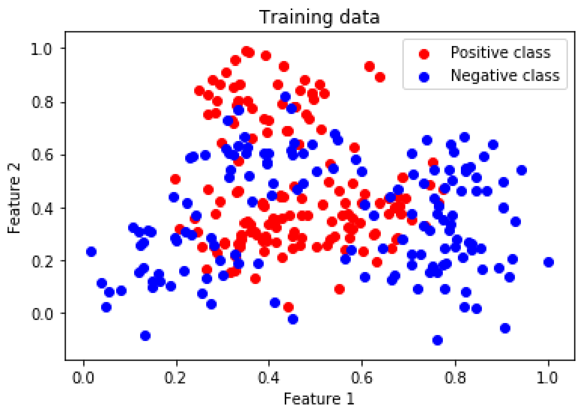
Imports

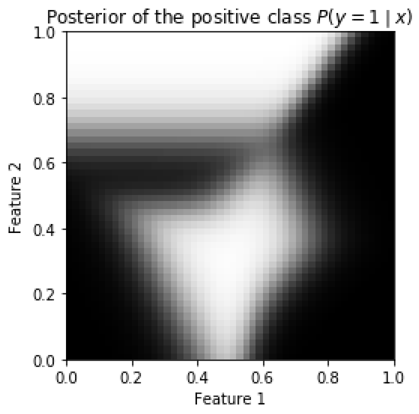
```
In [2]: import numpy as np
        import h5py
        import matplotlib.pyplot as plt
        from construct_data import construct_data
        from gradientChecking import checkNNGradients,compute_numerical_gradient
        from numpy.random import RandomState
        from numpy import unravel_index
```

Data Generation - Visualization

```
In [3]: prng=RandomState(1)


        %matplotlib inline
        plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plots
        plt.rcParams['image.interpolation'] = 'nearest'
        plt.rcParams['image.cmap'] = 'gray'

        %load_ext autoreload
        %autoreload 2

        features, labels, posterior = construct_data(300, 'train', 'nonlinear' , plusminus=Tru

        # Extract features for both classes
        features_pos = features[labels == 1]
        features_neg = features[labels != 1]

        # Display data
        fig = plt.figure(figsize=plt.figaspect(0.3))
        ax = fig.add_subplot(1, 2, 1)
        ax.scatter(features_pos[:, 0], features_pos[:, 1], c="red", label="Positive class")
        ax.scatter(features_neg[:, 0], features_neg[:, 1], c="blue", label="Negative class")

        ax.set_title("Training data")
        ax.set_xlabel("Feature 1")
        ax.set_ylabel("Feature 2")
        ax.legend()

        ax = fig.add_subplot(1, 2, 2)
        ax.imshow(posterior, extent=[0, 1, 0, 1], origin='lower')
        ax.set_title("Posterior of the positive class $P(y=1 \mid x)$")
        ax.set_xlabel("Feature 1")
        ax.set_ylabel("Feature 2")

        plt.show()


        #negative label values transformed from -1 to 0
        for i in range(labels.shape[0]):
            if labels[i]==-1:
                labels[i]=0
```

```
# creating alternative input
data=[]
for x,y in zip(features,labels): # creating alternative input
    x=x[:,np.newaxis]
    #v.append(np.array([x,y]))
    data.append(np.array([x,y]))
```





1.1 Analytical Dervative of a Function

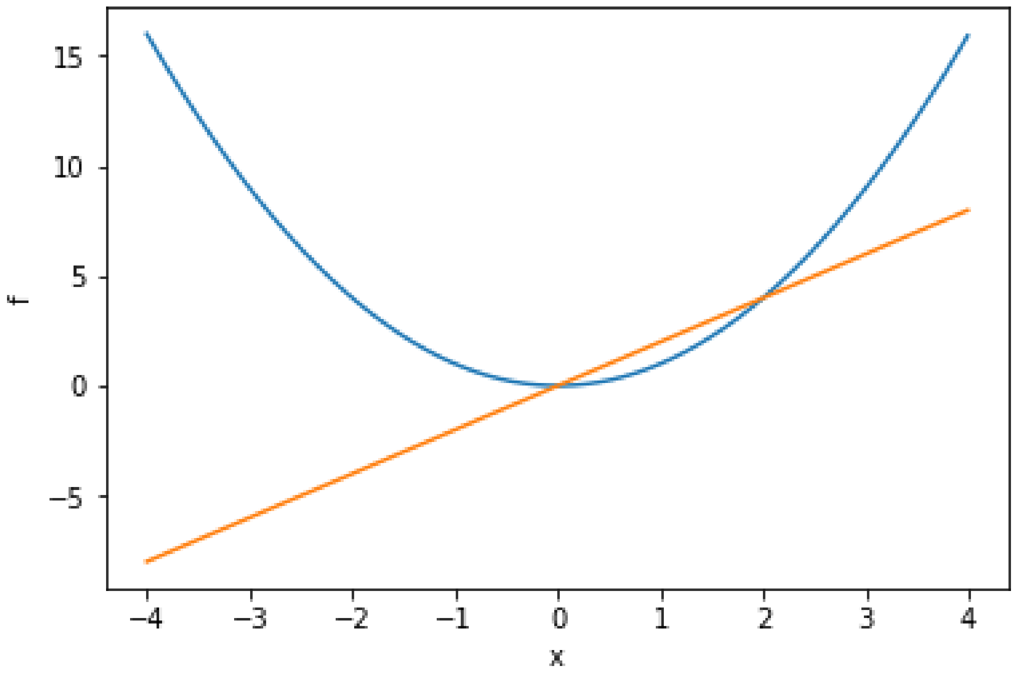
Here we present an example of how to compute the gradient of a function. Lets consider the function $f(x) = x^2$. Analytically we know that $f'(x) = 2x$. Thus for some x we can analytically caclulate the derivative of x. The following python functions compute f and f'.

```
In [4]: def square(x):
            f=x**2
            return f
        def derivative_square(x):
            f_dot=2*x
            return f_dot
```

We can visualize the function and its derivative using the following commands

```
In [5]: fig = plt.figure(figsize=plt.figaspect(0.3))
        ax = fig.add_subplot(1, 2, 1)
        x= np.arange(-4,4,0.01) #interval in which we plot the functions
        f=square(x)
        ax.plot(x,f)
        f_dot=derivative_square(x)
        ax.plot(x,f_dot)
        plt.xlabel('x')
        plt.ylabel('f')
```

```
Out[5]: <matplotlib.text.Text at 0x1a30f9448d0>
```



2 Question 1

2.1 Non-linear activation functions

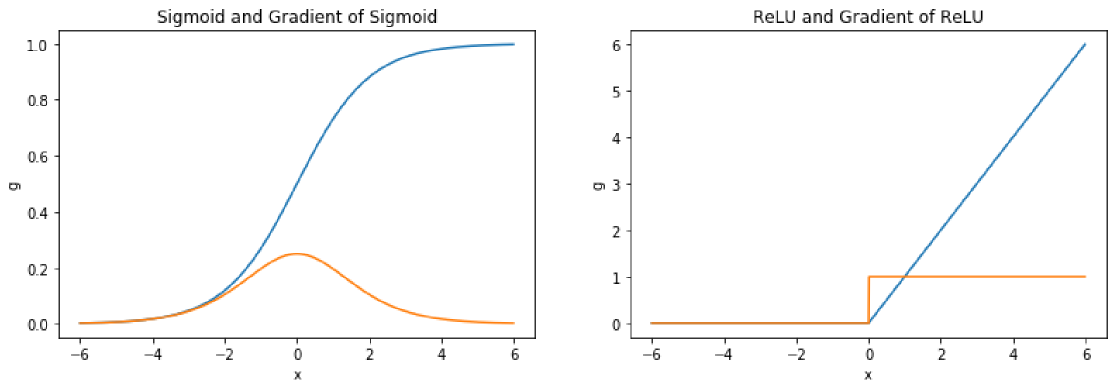
Fill in the functions that implement the non-linear functions which are used to produve the activations of the neurons of a neural network. Remember that for if $\sigma(x)$ is the sigmoid function, its derivative is $\sigma'(x) = \sigma(x) * (1 - \sigma(x))$. Finally if $relu(x) = max(0, x)$ then $relu'(x > 0) = 1$ and $relu'(x <= 0) = 0$. ##### DO NOT CHANGE THE NAMES OF THE FUNCTIONS AND THE ARGUMENTS.

```
In [6]: #TO DO (Q1)
        def sigmoid(z):
            return g

        def sigmoid_gradient(z):
            return g

        def relu(x):
            return g

        def relu_gradient(z):
            return g
        #TO DO (Q1)
```

Visualize the functions and their gradients by plotting their output in the [-6,6] interval:

```
In [7]: #########TO DO (Q1)###########




        #########TO DO (Q1)##########
```

```
Out[7]: <matplotlib.text.Text at 0x1a30fa55908>
```



3 Question 2

## The Neural Network In this assignement in order to define, train and utilize a neural network we are going to first use the implementation provided below. Every operation related to the network is defined as a **method** (ex. def feedforward , def backpropagation, etc) of the class **Neural_Network**. Try to go through the code to identify what each method inside the class does. Note that when first creating a Neural_Network class object (as shown here) , everything inside the def *init* method is executed. Also try to connect the notation inside the code with the notation from the slides

### Complete the feedforward method

You need to complete the feedforward method (Question 2a) which is defined inside the Neural_Network class (i.e in the code cell below). Then you need to complete the Cost_softmax method that should compute the cost when a softmax is used.

```
In [11]: class Neural_Network(object):
             # For Question 2 first focus on def __init__ and def feedforward
             def __init__(self, nnodes,activation_functions='sigmoid'):
                 # nnodes: number of hidden units per layer  - e.g. [2,5,10] indicates a three
                 self.num_layers = len(nnodes)
                 # weights, biases: list of the numpy arrays containing model parameters for l
                 # sampled originally from a gaussian distribution
                 self.sizes = nnodes
```

```
        prng=RandomState(2)
        self.biases = [prng.randn(y, 1) for y in nnodes[1:]]
        self.weights = [prng.randn(y, x) for x, y in zip(nnodes[:-1], nnodes[1:])]
        # non-linearity, specified when the network is initialized
        self.activation_functions=activation_functions
        self.costs=[]  # stores the costs during training
        self.accuracies=[] # stores the accuracies during training

    def feedforward(self, inputs):
        #Computes the output of the network if a is the input
        # In our case inputs.shape = (2,1), namely the input is a 2-dimensional vecto
        activations = []
        activations.append(inputs) # non linear outputs appended to this list
        zs= [] # linear outputs appended to this list

        # loop over all hidden layers (not the output layer ,which is handled after t
        for l in range(len(self.weights)-1):
            b = self.biases[l]
            w = self.weights[l]

            ##### TO DO Q2 linear output ##### 1 or 2 lines of code

            ##### TO DO Q2 linear output #####

            if self.activation_functions=='sigmoid':
            ##### TO DO Q2 sigmoid ##### 1 line of code

            ##### TO DO Q2 sigmoid #####

            elif self.activation_functions=='relu':

            ##### TO DO Q2 relu ##### 1 line of code

            ##### TO DO Q2 relu #####

            # save z values for backprop
            zs.append(z)
            # save activations for backprop
            activations.append(activation)
            # the input at the next layer is the activation of the previous layer
            inputs=activation

        # compute the output layer's linear and non-linear outputs

        w=self.weights[-1]
        b=self.biases[-1]
        z =np.dot(w,inputs)+b
        activation=sigmoid(z)
        activations.append(activation)
        zs.append(z)
        return activation,zs,activations


    def cost_function(self,X):
        # computes the cross entropy cost for a single example X=(x,y), where x is in
        cost = 0;
        for n in range(len(X)):
            x=X[n][0]
            y=X[n][1]
            posterior,dum1,dum2=self.feedforward(x) # (2,1)
            sample_cost  = -int(y==0) * np.log(posterior[0]) - int(1-y==0) * np.log(1
            sample_cost += -int(y==1) * np.log(posterior[1]) - int(1-y==1) * np.log(1
            cost = cost + sample_cost
        return cost

    def Gradient_Descent(self, training_data, epochs,lr,stop,test_data=None):
        # training_data is a list of tuples ([x1,x2],y) where y is the class and [x1,
        # lr is the learning_rate, it is scalar
        # epochs is a scalar value for the number of times the network is going to up
        for j in range(epochs):
            self.update_params(training_data,lr)
            cost =self.cost_function(training_data)
            self.costs.append(cost/len(training_data))
            if test_data:
                if test_data: n_test = len(test_data)
                correct = self.evaluate(test_data)
                print ("Epoch {0}: {1} / {2} Cost: {3}".format(j, correct, n_test,cos
                self.accuracies.append(correct/n_test)
                if correct > stop: return
            else:
                print ("Epoch {0} complete".format(j))

    def update_params(self, X, lr):
        # X is all the training data
        # X contains pairs (x,y) where x is the feature vector and y its class label
        N= len(X)
        grad_b = [np.zeros(b.shape) for b in self.biases]
        grad_w = [np.zeros(w.shape) for w in self.weights]
        #Gradient Descent parameter updates
        for x, y in X:
            delta_grad_b, delta_grad_w = self.backprop(x, y) # take an example backpr
            grad_b = [nb+dnb for nb, dnb in zip(grad_b, delta_grad_b)] # accumulate g
            grad_w = [nw+dnw for nw, dnw in zip(grad_w, delta_grad_w)] # accumulate g

        # Update the weights and biases using the gradient of the cost computed using
        # divide by N to scale the overall gradient by the number of training example
        self.weights = [w-lr*nw/N for w, nw in zip(self.weights, grad_w)]
        self.biases =  [b-lr*nb/N for b, nb in zip(self.biases,  grad_b)]



    def backprop(self, x, y):
        #Returns (grad_b, grad_w) representing the
        #gradients for the cost function  wrt b and w .
        grad_b = [np.zeros(b.shape) for b in self.biases]
        grad_w = [np.zeros(w.shape) for w in self.weights]

        ######### Forward pass of the backprop #########

        dummy,zs,activations= self.feedforward(x) #dummy is not used below

        # one hot encoding of the desired label
        # so if for example y=1 then the one-hot encoding of y is y =[0 , 1]

        if y==0:
            y=np.array([[1],[0]])
        else:
            y=np.array([[0],[1]])

        ######## Backward pass of the backprop ##########

        # Grad at the ouput layer
        z=zs[-1]
        delta = (activations[-1]- y)
        # grad wrt the parameters of the output layer

        grad_b[-1] = delta
        grad_w[-1] = np.dot(delta, activations[-2].T)

        # By uncommenting the following lines the you will be printing the shapes of
        # to compute the gradient at the output layer
        '''
        print('---- shapes')
        print('output-layer')
        print('activation',activations[-1].shape)
        print('z', z.shape)
        print('delta',delta.shape)
        print('grad_b ',grad_b[-1].shape,' grad_w ',grad_w[-1].shape,'delta',delta.sh
        '''

        ############# TO DO Q2b explain what is done by following for loop##########

        # Grad at the rest of the layers
        L = self.num_layers #  input layer + hidden layers + ouput layers
        # zs has L-1 elements , activations has L elements , self.weights has L-1 ele
        # start from 2nd-to-last layer (hence the L-2 index) and move backwards
        for l in range(L-2, 0,-1):
            # linear output of the l-th layer
            z = zs[l-1]

            # input to the l-th layer
            # which is the activation of the (l-1)-th layer
            activation=activations[l-1]

            # take the weights that map the output of the
            # l-th layer to the input of the (l+1)-th layer
            weights=self.weights[l]

            # compute the value of the gradient of the non-linearity of the l-th laye
            # given the z of the l-th layer
            if self.activation_functions=='relu': sg = relu_gradient(z)
            elif self.activation_functions=='sigmoid': sg = sigmoid_gradient(z)

            # backpropagate delta from (l+1)-th layer to l-th layer
            prod  = np.dot(weights.T, delta)
            delta = prod * sg

            # gradients of the parameters of the l-th layer
            grad_b[l-1] = delta
            grad_w[l-1] = np.dot(delta, activation.T)

            # By uncommenting the following lines the
            # function will be printing the shapes of each variable

            '''
            print('---- shapes')
            print('layer:',l)
            print('activation ',activation.shape)
            print('z ', z.shape)
            print('w ',weights.shape)
            print('prod ',prod.shape,'sg',sg.shape)
            print('delta ',delta.shape)
            print('grad_b ',grad_b[-l].shape,' grad_w ',grad_w[-l].shape)
            '''
        return (grad_b, grad_w)

        ############ TO DO Q2b explain what is done by following for loop##########

    def evaluate(self, data):
        correct = 0
        for index in range(len(data)):
            feature = data[index][0]
            label   =  data[index][1]
            output,dum1,dum2 = self.feedforward(feature)
            correct = correct  + int(label == np.argmax(output))
        return correct
```

#### Gradient checking: We are going to use *gradient checking* to check if the numerical (using approximations) and analytical (computed by our backprop implementation) gradient computations match. To test that we are going to check the gradients computed for a small neural network.

Suppose we have a neural network with 2 dimensional inputs , 2 hidde layers of 4 neurons each with sigmoid non-linearities and an output layer of 2 neurons with sigmoid non-linearity. This network is initialized as shown by the code in the next cell.

```
In [15]: nnodes2=[2,4,2]
         activation_functions_checking='sigmoid'
         nnet_checking = Neural_Network(nnodes2,activation_functions_checking)
```

We are going to use a few training examples to check if the analytical and numerical gradients match.

```
In [16]: m=50 # using 50 examples to do gradient checking
         train_data = data[1:m]
         checkNNGradients(nnet_checking,train_data)                 # compute_numerical_gradie

Relative difference 1.24765069294e-07 for layer 0 parameters
Analytical and numerical gradients match
as relative distance is less that 1e-5
Relative difference 2.5264340071e-08 for layer 1 parameters
Analytical and numerical gradients match
as relative distance is less that 1e-5
```

#### Define a Network To initialize a Neural Network we must provide two arguments: *nnodes* and activation_functions. *nnodes* is a list. Each element of *nnodes* defines the number of neurons of each layer of the network. By convention the first element of *nnodes* defines the input layer of the network that is not associated with any weights and biases. The rest of the elements of *nnodes* are used to initialize the weigths and biases of the hidden layers and the output layer. The activation_functions argument is a string that defines the type of non-linear function to be used for each neuron of the hidden layer. Our implementation supports ReLUs and sigmoids as activation functions. The output layer is set to always be a sigmoid function.

Execute the following three cells to 1) define a network 2) train it 3) visualize the estimated posterior

```
In [14]: nnodes=[2, 10,10, 2]
         activation_functions='sigmoid' # use 'sigmoid' or 'relu'
         nnet = Neural_Network(nnodes,activation_functions)
         #print(nnet.weights) # uncomment to see weights after they have been initialized
```

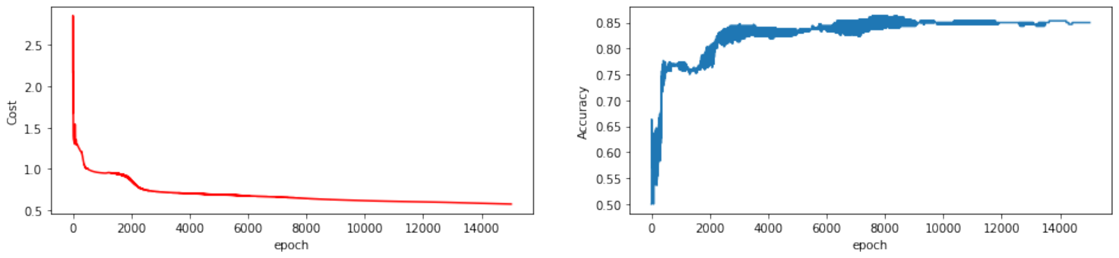
```
In [ ]: learning_rate=2 #
        epochs = 15000  ## usually has already converged around that number
        stop = 260 #  number of correctly classified performance to which we wish to stop trai
        train_data = data
        test_data = data #evaluate the performance on the train data
        nnet.Gradient_Descent(train_data,epochs,learning_rate,stop,test_data)
```

3.0.1 Train a network with sigmoid non-linearities across all layers

To train the Network we use the Gradient_Descent method defined inside the Neural_Network Class

```
In [353]: #### RUN this after training is done
          #access the cost and accuracy during training by referencing the nnet object's attri
          #This is done in the following way:
          cost_training= nnet.costs
          accuracy_training=nnet.accuracies
          # Plots the cost and accuracy evolution during training
          fig = plt.figure(figsize=plt.figaspect(0.2))
          ax1 = fig.add_subplot(1, 2, 1)
          ax1.plot(cost_training,'r')
          plt.xlabel('epoch')
          plt.ylabel('Cost')
          ax1 = fig.add_subplot(1, 2, 2)
          ax1.plot(accuracy_training)
          plt.xlabel('epoch')
          plt.ylabel('Accuracy')
          A=np.array(accuracy_training)
          best_epoch=np.argmax(A)
          print('best_accuracy:',max(accuracy_training),'achieved at epoch:',best_epoch)
```

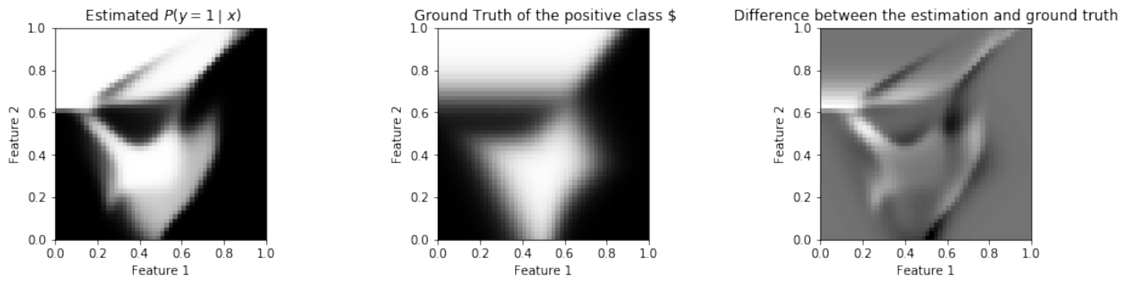
```
best_accuracy: 0.8633333333333333 achieved at epoch: 7576
```



Visualize the posterior of the network trained above

```
In [330]: #### RUN this after training is done
          x_rng = y_rng = np.linspace(0, 1, 50)
```

```
        gridx,gridy = np.meshgrid(x_rng, y_rng)
        p1=np.zeros((50,50))
        p0=np.zeros((50,50))
        for i in range(50):
            for j in range(50):
                v = np.array([gridx[i,j],gridy[i,j]])
                v=v[:,np.newaxis]
                out,dum,dum2=nnet.feedforward(v)
                p0[i,j]=out[0]
                p1[i,j]=out[1]

        fig = plt.figure(figsize=plt.figaspect(0.2))
        ax1 = fig.add_subplot(1, 3, 1)
        ax1.imshow(p1, extent=[0, 1, 0, 1], origin='lower')
        ax1.set_title(" Estimated $P(y=1 \mid x)$")
        ax1.set_xlabel("Feature 1")
        ax1.set_ylabel("Feature 2")

        ax1 = fig.add_subplot(1, 3, 2)
        ax1.imshow(posterior, extent=[0, 1, 0, 1], origin='lower')
        ax1.set_title(" Ground Truth of the positive class $")
        ax1.set_xlabel("Feature 1")
        ax1.set_ylabel("Feature 2")

        ax1 = fig.add_subplot(1, 3, 3)
        ax1.imshow(p1-posterior, extent=[0, 1, 0, 1], origin='lower')
        ax1.set_title("Difference between the estimation and ground truth")
        ax1.set_xlabel("Feature 1")
        ax1.set_ylabel("Feature 2")
        plt.show()
```



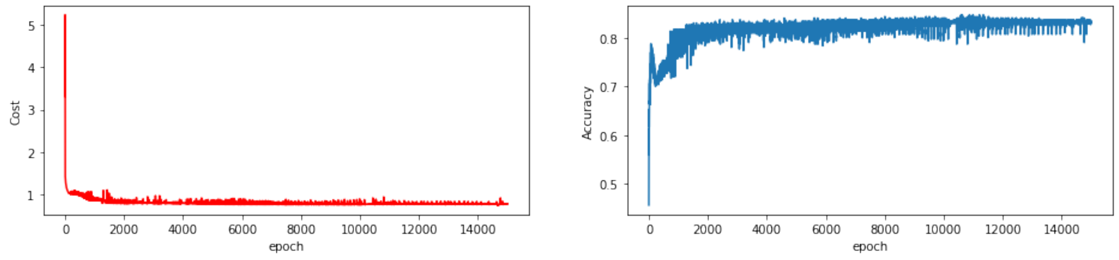
#### Train a network with Relu non-linearities for the hidden layers and sigmoid non-linearity for the output layer __ Use nnodes = [2 , 10 , 10 ,2]__

```
In [ ]: nnodes=[2, 10,10, 2]
        # Recommended to set the learning rate less than or equal to 0.2
        # use the previous network that was trained above as an example
```

```
############ TO DO Q2d ############################
#nnet_2 = # Name the network nnet_2 in order to avoid erasing the previous model!



###########  TO DO Q2d ###########################
```

```
In [339]: #### RUN this after training is done
          #access the cost and accuracy during training by referencing the nnet object's attri
          #This is done in the following way:
          cost_training= nnet_2.costs
          accuracy_training=nnet_2.accuracies
          # Plots the cost and accuracy evolution during training
          fig = plt.figure(figsize=plt.figaspect(0.2))
          ax1 = fig.add_subplot(1, 2, 1)
          ax1.plot(cost_training,'r')
          plt.xlabel('epoch')
          plt.ylabel('Cost')
          ax1 = fig.add_subplot(1, 2, 2)
          ax1.plot(accuracy_training)
          plt.xlabel('epoch')
          plt.ylabel('Accuracy')
          A=np.array(accuracy_training)
          best_epoch=np.argmax(A)
          print('best_accuracy:',max(accuracy_training),'achieved at epoch:',best_epoch)
```

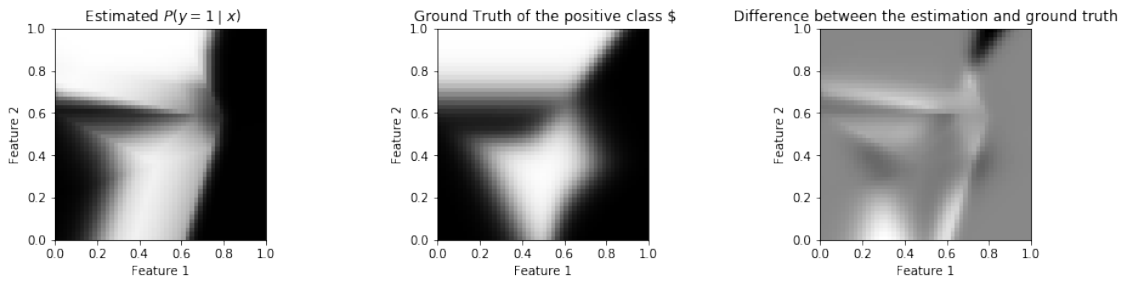
```
best_accuracy: 0.8466666666666667 achieved at epoch: 10591
```



Visualize the posterior of the model trained above

```
In [340]: #### RUN this after training is done
          x_rng = y_rng = np.linspace(0, 1, 50)
          gridx,gridy = np.meshgrid(x_rng, y_rng)
          p1=np.zeros((50,50))
          p0=np.zeros((50,50))
          for i in range(50):
```

```
        for j in range(50):
            v = np.array([gridx[i,j],gridy[i,j]])
            v=v[:,np.newaxis]
            out,dum1,dum2=nnet_2.feedforward(v)
            #print(out[0],out[1])
            p0[i,j]=out[0]
            p1[i,j]=out[1]
            #print(p0[i,j],p1[i,j])

fig = plt.figure(figsize=plt.figaspect(0.2))
ax1 = fig.add_subplot(1, 3, 1)
ax1.imshow(p1, extent=[0, 1, 0, 1], origin='lower')
ax1.set_title(" Estimated $P(y=1 \mid x)$")
ax1.set_xlabel("Feature 1")
ax1.set_ylabel("Feature 2")

ax1 = fig.add_subplot(1, 3, 2)
ax1.imshow(posterior, extent=[0, 1, 0, 1], origin='lower')
ax1.set_title(" Ground Truth of the positive class $")
ax1.set_xlabel("Feature 1")
ax1.set_ylabel("Feature 2")

ax1 = fig.add_subplot(1, 3, 3)
ax1.imshow(p1-posterior, extent=[0, 1, 0, 1], origin='lower')
ax1.set_title("Difference between the estimation and ground truth")
ax1.set_xlabel("Feature 1")
ax1.set_ylabel("Feature 2")
plt.show()
```


Estimated $P(y=1 \mid x)$
Ground Truth of the positive class $
Difference between the estimation and ground truth
Feature 1
Feature 2