Neural Network Classifier

In this assignment, your task is to train a type of Neural network classifier on the synthetic dataset that was also used in previous assignements. Please read the assignment entirely before you start coding. Most of the code that implements the neural network functionalities are provided to you. What you are expected to do is fill in certain missing parts that are required and then train two different networks and visualize the estimated posterior.

- Question 1 (10%) Fill in the code to implement the `sigmoid` and `relu` functions and their derivatives. Then, plot all four functions to verify that your implementation is correct. See the provided example first.
- Question 2 (80%) Fill in the code inside the `NeuralNetwork` class where necessary.
  - a. First, complete the code in the `feedforward` method of the `NeuralNetwork` class. Assuming that the input of the l-th layer of the network is a^{l-1} (a^{l-1} can be the non-linear output or activation of a previous layer or initially the input feature vector) then the linear-output of the layer is:
  $$z^l = Wa^{l-1} + b$$
  Then the non-linear output or "activation" is $a^l = f(z^l)$ where $f(\cdot)$ is the non-linear activation function of the layer, applied element-wise on z^l. In our case, $f(\cdot)$ is either a `sigmoid` or `relu` function.
  - b. In the `feedforward` method of the `NeuralNetwork` class, write the code that implements the softmax function (see pages 67, 73 of the course slides). If z^L is the linear-output of the output layer of the network then applying the softmax function on each of its elements z_k^L is described by:
  $$softmax(z_k) = \frac{\exp(z_k^L)}{\sum_k \exp(z_k^L)}, k = 0, \ldots, C-1$$
  Note: z^L will be a 2-dimensional vector as our synthetic dataset has $C = 2$ classes. Consequently $softmax(z_0^L) = P_{estimated}(y = 0 \mid x)$ and $softmax(z_1^L) = P_{estimated}(y = 1 \mid x)$.
  - c. Fill in the `loss_function` method of the `NeuralNetwork` class so that it computes the value of the loss used to train the network (see page 68 of the coures slides, where it is refered to as "optimization criterion"):
  $$L(W) = -\sum_{i=1}^{N}\sum_{c=0}^{1} y_c^i \log(g_c(x^i; W))$$
  where i indexes training examples, c indexes the two classes {0,1} of our synthetic data, W is the set of weights of the network and g_c is the posterior of class c as computed by the network. As implied in pages 68 and 70 of the course slides, for y^i being the true class label of example x^i, y_c^i is the one-hot label encoding of y^i, meaning that $y_c^i = 1$ if $y^i = c$ and $y_c^i = 0$ if $y^i \neq c$. Use the gradient checking code (found here) to ensure all your additions to the `NeuralNetwork` class are working correctly.
  - d. Go through and try to understand the `backprop` method of the `Neural Network` class which is provided. Briefly explain what is done by the for loop at the end of the `backprop` method. Add your explanation in a markdown cell which here.
- Question 3 (10%) Use the `NeuralNetwork` class to define and train two different neural networks.
  - a. Train a network with a sigmoid non-linear activation function. Then plot the accuracy and error curves. Finally, visualize the estimated posterior. The code needed is provided.
  - b. By using the previous question's code as an example, train a network with relu non-linear activation functions. Then with the provided code, plot the accuracy and error curves and visualize the estimated posterior. Briefly compare the performance of the two networks in a markdown cell which must be added at the end of this notebook here.

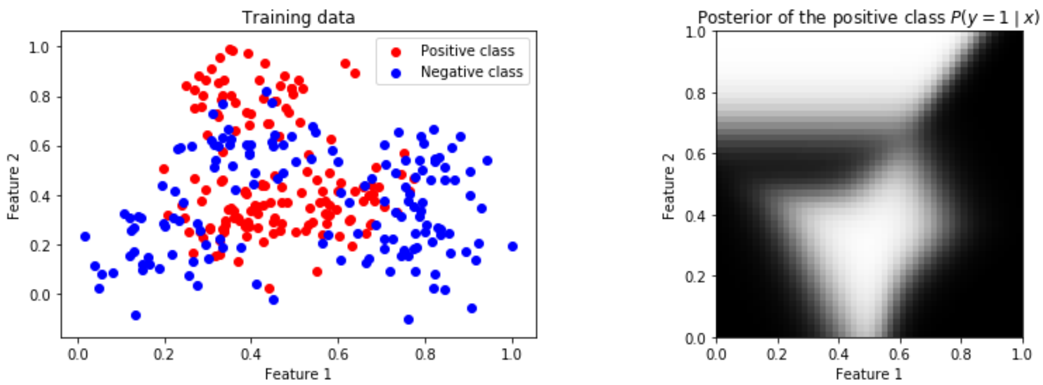
Code

Imports

In [1]:
```
import h5py
import matplotlib.pyplot as plt
import numpy as np
import pickle
from numpy.random import RandomState
from construct_data import construct_data
from gradientChecking import checkNNGradients, compute_numerical_gradien
t
```

Data Generation - Visualization

In [2]:
```
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'

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

# 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="Posit
ive class")
ax.scatter(features_neg[:, 0], features_neg[:, 1], c="blue", label="Nega
tive 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()

# Our dataset arrays are modified to have a more convenient format for t
his assignment
# instead of two arrays features (300,2) and labels (300,)
# we now have a list of 300 elements called data
# each element of data is a tuple (x,y)
# where x is  a (2,1) feature vector and y is a scalar label with value
either 0 or 1
data = []
for x, y in zip(features, labels): # creating alternative input
    x = x[:, np.newaxis]
    data.append(np.array([x,y]))
```



Helper function for visualizing the posterior

```
In [58]: def visualize_posterior(nnet):
    x_rng = y_rng = np.linspace(0, 1, 50)
    gridx, gridy = np.meshgrid(x_rng, y_rng)
    p_estimated_class1 = 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, _, _ = nnet.feedforward(v)
            p_estimated_class1[i, j] = out[1]

    fig = plt.figure(figsize=plt.figaspect(0.2))
    ax1 = fig.add_subplot(1, 3, 1)
    ax1.imshow(p_estimated_class1, 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(posterior-p_estimated_class1+1, extent=[0, 1, 0, 1], orig
in='lower')
    ax1.set_title("Difference between the ground truth and estimated pos
terior")
    ax1.set_xlabel("Feature 1")
    ax1.set_ylabel("Feature 2")
    plt.show()

def visualize_cost_accuracy_curves(accuracy_training, cost_training):
    # 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')
    ax1.set_xlabel('epoch')
    ax1.set_ylabel('Cost')
    ax2 = fig.add_subplot(1, 2, 2)
    ax2.plot(accuracy_training)
    ax2.set_xlabel('epoch')
    ax2.set_ylabel('Accuracy')
    A = np.array(accuracy_training)
    best_epoch = np.argmax(A)
    print('best_accuracy:', max(accuracy_training), 'achieved at epoc
h:', best_epoch)
```

Analytical Derivative of a Function

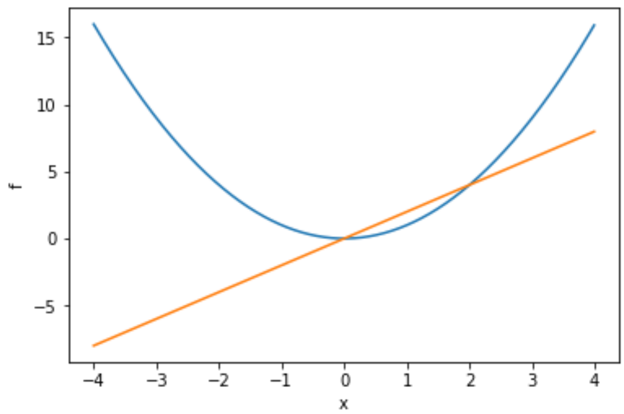
Here we present an example of how we can define (and visualize) each nonlinear function and its derivative. Let us consider the function $f(x) = x^2$. Analytically we know that $f'(x) = 2x$. Thus for some x we can analytically calculate the derivative of x. The following python functions compute f and f'.

```
In [59]: def square(x):
    f = x**2
    return f

def square_gradient(x):
    f_prime = 2*x
    return f_prime
```

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

In [60]:
```
fig = plt.figure(figsize=plt.figaspect(0.3))
ax = fig.add_subplot(1, 2, 1)
x = np.arange(-4, 4, 0.01)
f = square(x)
ax.plot(x, f)
f_prime = square_gradient(x)
ax.plot(x, f_prime)
ax.set_xlabel('x')
ax.set_ylabel('f')
```

Out[60]: Text(0, 0.5, 'f')



Question 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 $relu'(x <= 0) = 0$.

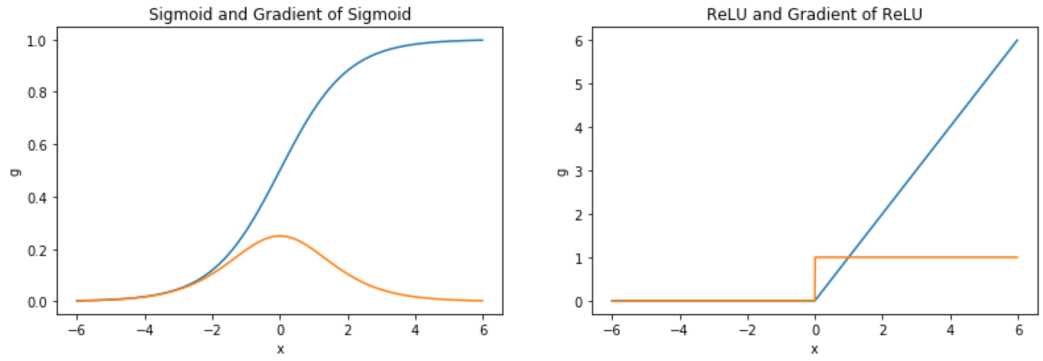
Note: Do not change the functions and their arguments!

In [61]:
```
# TO DO (Q1)
def sigmoid(z):

    return g

def sigmoid_gradient(z):

    return g
# /TO DO (Q1)

# TO DO (Q1)
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 [62]:
```
#TO DO (Q1)


# /TO DO (Q1)
```

Out[62]: Text(0, 0.5, 'g')



Question 2

The Neural Network Class

In this assignement, in order to define, train and use a neural network you are first required to complete the implementation provided below. Every operation related to the network is defined as a method (e.g. `feedforward` , `backpropagation` , etc.) of the `NeuralNetwork` class. You only need to complete the code inside the `feedforward` and `loss_function` methods.

Go through the code to identify what each method inside the class does. Note that when first creating a `NeuralNetwork` class instance (for example as [here]), the `__init__` method is executed and randomly initializes the network's weights, which then constitute an attribute of that instance. These weights are then modified by calling the `gradient_descent` method used to train the network.

In [63]:

```python
class NeuralNetwork(object):
    # For Question 2 first focus on __init__ and feedforward
    def __init__(self, nnodes, activation_functions='sigmoid'):
        # nnodes: number of hidden units per layer  - e.g. [2,5,10] indicates a three-layer network with the respective number of nodes
        self.num_layers = len(nnodes)
        # weights, biases: list of the numpy arrays containing model parameters for linear layers
        # 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.losses = []  # stores the losses during training
        self.accuracies = []  # stores the accuracies during training
        self.epoch_best = 0 # epoch during training when accuracy was max

    def feedforward(self, x):
        # Computes and returns the output of the network for a given input x
        # In our case x.shape = (2,1), i.e. the input is a 2-dimensional vector
        # It also returns intermediate linear and non-linear outputs of the layers
        # that can are for backpropagation when traning
        activations = [] # here we store the non-linear outputs of layers
        activations.append(x)
        zs = [] # here we store linear-outputs of layers

        # feedforward computation through all layers except the output layer
        for l in range(len(self.weights)-1):
            b = self.biases[l]
            w = self.weights[l]

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


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

            if self.activation_functions == 'sigmoid':
            ##### TO DO Q2 sigmoid #####


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

            elif self.activation_functions == 'relu':

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


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

            # save z values for backprop (linear output of each layer)
            zs.append(z)
            # save activations for backprop (non-linear output of each layer)
            activations.append(activation)
            # the input at the next layer is the activation of the previous layer
            x = activation

        # feedforward computation through the output layer
```

Explanation of the backprop for loop

Answer:

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 hidden 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 [64]:
```
nnodes2 = [2, 4, 4, 2]
activation_functions2 = 'sigmoid'
nnet_checking = NeuralNetwork(nnodes2, activation_functions2)
```

We are going to use a few training examples (m=50) to check if the analytical and numerical gradients match.

In [65]:
```
m = 50 # using 50 examples to do gradient checking
train_data = data[1:m]
checkNNGradients(nnet_checking, train_data)  # compute_numerical_gradien
t
```

```
Relative difference 2.7629529695019207e-07 for layer 0 parameters
Analytical and numerical gradients match
as relative distance is less than 1e-5
Relative difference 5.717020591385936e-07 for layer 1 parameters
Analytical and numerical gradients match
as relative distance is less than 1e-5
Relative difference 5.076150079980013e-07 for layer 2 parameters
Analytical and numerical gradients match
as relative distance is less than 1e-5
```

Question 3

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 dimensionality of the input layer of the network that is not associated with any weights and biases (i.e if the input to a network is $x \in R^2$ then the dimensionality of the input layer is 2). 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 relu and sigmoid as non-linear activation functions.

```
In [66]: nnodes = [2, 10, 10, 2]
activation_functions = 'sigmoid' # this can either be 'sigmoid' or 'relu
'
nnet = NeuralNetwork(nnodes, activation_functions) # creating a NeuralNe
twork instance
# accessing the weights attribute of the nnet instance we can print weig
ht shapes per layer
print(['layer {}: w {}'.format(layer, w.shape) for w, layer in zip(nnet.
weights, range(len(nnet.weights)))])
```

```
['layer 0: w (10, 2)', 'layer 1: w (10, 10)', 'layer 2: w (2, 10)']
```

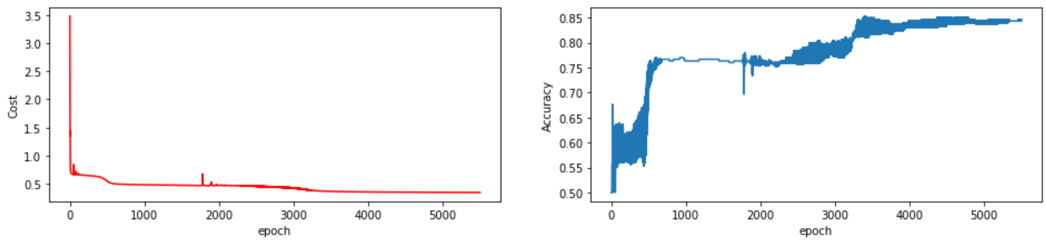
Train a network with sigmoid activation functions.

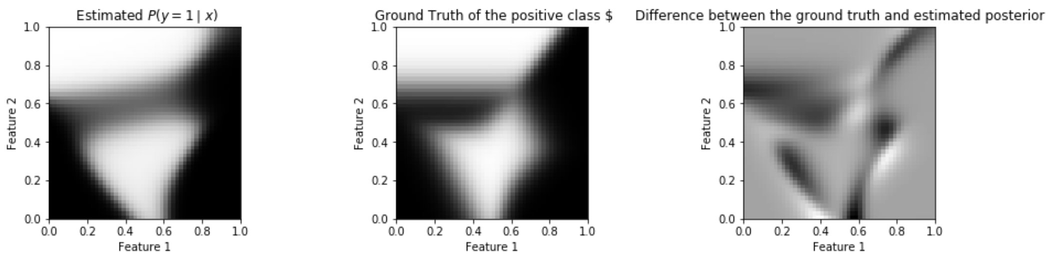
Note that for each training epoch we compute the mean cost and accuracy on the training set. Once the training is completed visualize the cost / accuracy curves and the estimated posterior using the provided code. It is recomended that you keep the learning rate and number of epochs as provided below.

```
In [ ]: learning_rate = 2
epochs = 5000  # usually has converged for that many epochs
train_data = data
test_data = data
nnet.gradient_descent(train_data, epochs, learning_rate, test_data)
```

```
In [72]: # We get the cost and accuracy during training
# by referencing the nnet object's attributes costs and accuracies which
we give to the visualization function
losses_training = nnet.losses
accuracy_training = nnet.accuracies
visualize_cost_accuracy_curves(accuracy_training, losses_training)
# we give the nnet object to the second visualization function to visual
ize the estimated posterior
visualize_posterior(nnet)
```

```
best_accuracy: 0.8533333333333334 achieved at epoch: 3396
```





Train a network with ReLU activation functions.

Define a new network with the same number of nodes and name it nnet_2 in order to avoid erasing the previous model. Once the training is completed visualize the cost / accuracy curves and the estimated posterior using the provided code.

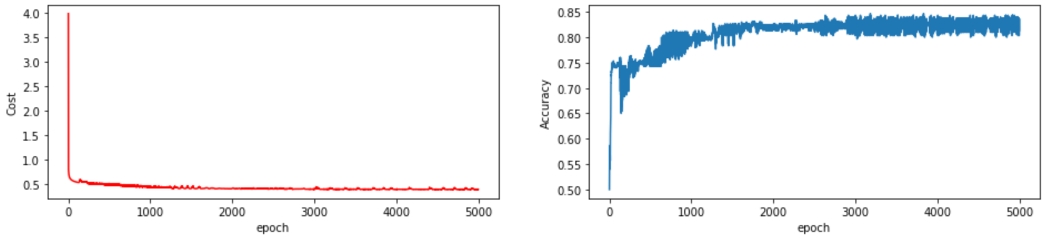
In [ ]:

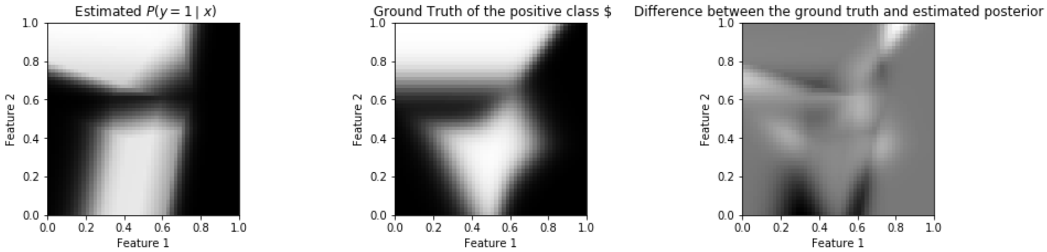
```python
nnodes = [2, 10, 10, 2]
learning_rate = 0.2
epochs = 5000
############ TO DO Q2d ############################






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

In [74]:

```python
losses_training = nnet_2.losses
accuracy_training = nnet_2.accuracies
visualize_cost_accuracy_curves(accuracy_training, losses_training)
visualize_posterior(nnet_2)
```

```
best_accuracy: 0.8466666666666667 achieved at epoch: 3182
```





Compare the two networks below:

Answer:

In [ ]: