Training a Neural Network with PyTorch (60%)

In this assignment, your task is to train a Neural Network or Multilayered Perceptron (MLP) classifier on synthetic data using PyTorch. We'll start by creating the same network architecture as in last week's assignment, only using PyTorch instead of our own implementation. Before you start, make sure you've installed PyTorch in your conda installation.

You may do this using Anaconda navigator: select Environments -> base root. Select "all" from the dropdown. Search for "pytorch". Select the checkbox for pytorch and click "apply". Now do the same for "torchvision".

Alternatively, you may do this on the command line: `conda install pytorch torchvision`

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 15%
- Question 2 15%
- Question 3 15%
- Question 4 15%

Imports

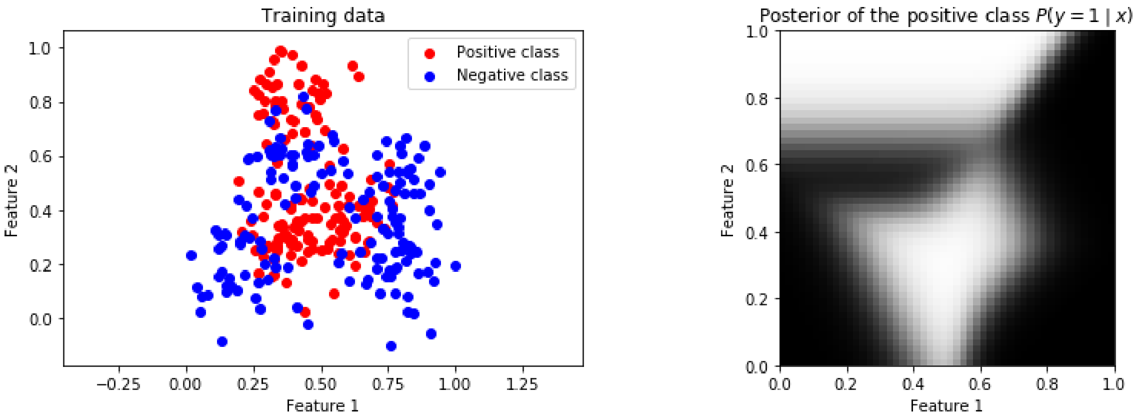
In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
from construct_data import construct_data
from numpy.random import RandomState
from numpy import unravel_index


import torch
#Set torch seed
torch.manual_seed(42)

import torchvision
import torchvision.transforms as transforms

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

import time
```

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'

training_features, training_labels, posterior = construct_data(300, 'train', 'no
nlinear' , plusminus=False)

# Extract features for both classes
features_pos = training_features[training_labels == 1]
features_neg = training_features[training_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 clas
s")
ax.scatter(features_neg[:, 0], features_neg[:, 1], c="blue", label="Negative cla
ss")
ax.axis('equal')
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()
```



Generate test set

In [3]:

```
test_features, test_labels, _ = construct_data(100, 'test', 'nonlinear' , plusmi
nus=False)
```

Convert data to Torch tensors

All computations in torch are performed on tensors (generalizations of a matrix that can be indexed in more than 2 dimensions). Therefore we need to first convert our testing and training data to tensors.

In [4]:

```
training_features_tensor = torch.tensor(training_features, dtype=torch.float)
training_labels_tensor = torch.tensor(training_labels, dtype=torch.float)
test_features_tensor = torch.tensor(test_features, requires_grad=False, dtype=to
rch.float)
test_labels_tensor = torch.tensor(test_labels, requires_grad=False, dtype=torch.
float)
```

Next we'll use the training label and feature tensors to create a TensorDataset as well as a DataLoader that allows us to load batches of our data during each training iteration.

In [5]:

```
# loader for training set
batch_size = 40
trainset = torch.utils.data.TensorDataset(training_features_tensor, training_lab
els_tensor)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuff
le=True, num_workers=2)
```

Simple Network in PyTorch

The following cell, demonstrates how to create a very simple network with only one hidded layer with 3 nodes. In the constructor function `__init__()` we create the layers and within function `forward()` we define the structure of the network.

In [2]:

```
class SimpleNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(2, 3)
        self.fc2 = nn.Linear(3, 2)

    def forward(self, x):
        x = self.fc1(x)
        x = torch.sigmoid(x)
        x = self.fc2(x)
        return x
```

Train a network with sigmoid non-linearities across all layers

In this assignment we'll create the same neural network we created last week, with [2,10,10,2] nodes (i.e. 2 inputs, 2 layers with 10 nodes and 2 outputs). We'll use sigmoid activations for the hidden layers. There should be no activation on the final layer.

- Question 1: Use the previous simple network as a guide in order to create the network described. Add the necessary lines inside `__init__` and `forward`.

In [7]:

```
class SigmoidNet(nn.Module):
    def __init__(self):
        super().__init__()
        ###### TODO QUESTION 1 ########
        self.fc1 = 
        self.fc2 = 
        self.fc3 = 
        ###### TODO QUESTION 1 ########
    def forward(self, x):
        ###### TODO QUESTION 1 ########

        ###### TODO QUESTION 1 ########
        return x
```

In [8]:

```
#Next we'll create an instance of our SigmoidNet class
sigmoid_net = SigmoidNet()
```

In [9]:

```
# Here we define the learning rate and the number of epochs for our training.
#Hint: start with a small number of epochs (around 500) to check if everything
# is working and only then run it for 4000 epochs)
learning_rate=0.05 #
epochs = 4000  ## usually has already converged around that number

## Here we pick the cost function and the optimizer for our training
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(sigmoid_net.parameters(), lr=learning_rate, momentum=0.9)
```

Training SigmoidNet

In the next cell the actual training happens. The network is trained for many epochs and for each of the epochs we split our training data into mini batches using the DataLoader that we created previously and iteratively feed them into our network. After each iteration/optimizer step, the parameters are updated in such a way that the loss is minimized.

In [10]:

```
start = time.time()

sigmoid_costs=[]
sigmoid_accuracies = []

for epoch in range(epochs):  # loop over the dataset multiple times

    running_loss = 0.0
    num_of_batches = 0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        logits = sigmoid_net(inputs)
        loss = criterion(logits, labels.type(torch.LongTensor))

        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        num_of_batches +=1

    # calculate test accuracy
    test_logits = sigmoid_net(test_features_tensor)
    _, test_predictions = torch.max(test_logits.data, 1)
    correct_predictions = (test_predictions.int() == test_labels_tensor.int()).s
um().numpy()
    test_length = test_labels_tensor.size()[0]
    accuracy = correct_predictions/test_length
    if epoch%(epochs//20) == 0:
        print('Epoch: %d/%d,  loss: %.3f, test accuracy: %.3f'%(epoch + 1,epochs
, running_loss / num_of_batches, accuracy))
    sigmoid_costs.append(running_loss/num_of_batches)
    sigmoid_accuracies.append(accuracy)



end = time.time()
duration = end-start
print('Finished Training in %d seconds'%(end-start))
```

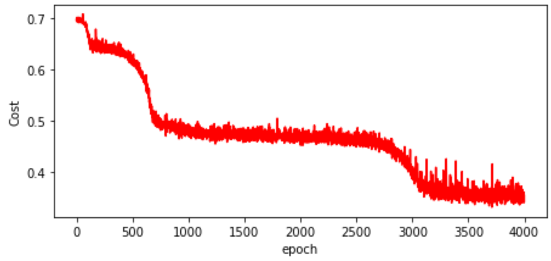
```
Epoch: 1/4000,  loss: 0.698, test accuracy: 0.500
Epoch: 201/4000,  loss: 0.644, test accuracy: 0.590
Epoch: 401/4000,  loss: 0.635, test accuracy: 0.620
Epoch: 601/4000,  loss: 0.583, test accuracy: 0.680
Epoch: 801/4000,  loss: 0.505, test accuracy: 0.810
Epoch: 1001/4000,  loss: 0.482, test accuracy: 0.820
Epoch: 1201/4000,  loss: 0.483, test accuracy: 0.820
Epoch: 1401/4000,  loss: 0.475, test accuracy: 0.810
Epoch: 1601/4000,  loss: 0.473, test accuracy: 0.800
Epoch: 1801/4000,  loss: 0.467, test accuracy: 0.810
Epoch: 2001/4000,  loss: 0.471, test accuracy: 0.820
Epoch: 2201/4000,  loss: 0.471, test accuracy: 0.810
Epoch: 2401/4000,  loss: 0.463, test accuracy: 0.830
Epoch: 2601/4000,  loss: 0.459, test accuracy: 0.830
Epoch: 2801/4000,  loss: 0.451, test accuracy: 0.840
Epoch: 3001/4000,  loss: 0.399, test accuracy: 0.880
Epoch: 3201/4000,  loss: 0.362, test accuracy: 0.850
Epoch: 3401/4000,  loss: 0.356, test accuracy: 0.880
Epoch: 3601/4000,  loss: 0.354, test accuracy: 0.870
Epoch: 3801/4000,  loss: 0.364, test accuracy: 0.860
Finished Training in 109 seconds
```

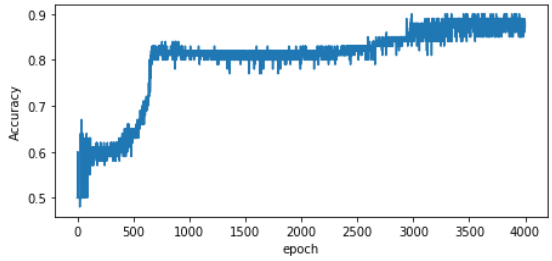
In [11]:

```python
def plot_cost_accuracy(costs, 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(costs,'r')
    plt.xlabel('epoch')
    plt.ylabel('Cost')
    ax1 = fig.add_subplot(1, 2, 2)
    ax1.plot(accuracies)
    plt.xlabel('epoch')
    plt.ylabel('Accuracy')
    plt.show()
    A=np.array(accuracies)
    best_epoch=np.argmax(A)
    best_accuracy = max(accuracies)
    print('best_accuracy:',best_accuracy,'achieved at epoch:',best_epoch)

    return best_accuracy
```

In [12]:

```python
plot_cost_accuracy(sigmoid_costs, sigmoid_accuracies)
```






```
best_accuracy: 0.9 achieved at epoch: 2988
```

Out[12]:

```
0.9
```

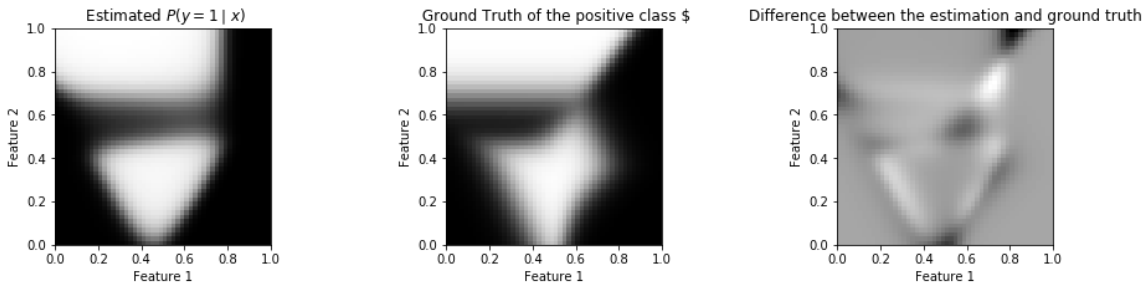
Visualize the posterior of the network trained above

In [13]:

```
def visualize_posterior(gt_posterior, network):
    #### RUN this after training is done
    x_rng = y_rng = np.linspace(0, 1, 50)
    gridx,gridy = np.meshgrid(x_rng, y_rng)

    grid_features = np.concatenate((gridx.reshape((-1,1)), gridy.reshape((-1,1
))),axis=1)
    grid_tensor = torch.tensor(grid_features, dtype=torch.float)
    logits = network(grid_tensor)
    soft_max = torch.nn.Softmax(dim=1)
    softmax_outputs = soft_max(logits)

    posterior_0 = softmax_outputs[:,0].data.numpy()
    posterior_1 = softmax_outputs[:,1].data.numpy()
    posterior_0 = posterior_0.reshape(50,50)
    posterior_1 = posterior_1.reshape(50,50)


    fig = plt.figure(figsize=plt.figaspect(0.2))
    ax1 = fig.add_subplot(1, 3, 1)
    ax1.imshow(posterior_1, 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(gt_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_1-gt_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()
```

In [14]:

```
visualize_posterior(posterior,sigmoid_net)
```



Train a network with ReLU non-linearities across all layers

In this assignment we'll create the same neural network as previously, but we'll use relu activations for the hidden layers instead. (Note: relu activations can be called as `torch.relu()`. As before, there should be no activation on the final layer.

- Question 2: Add the necessary lines inside `__init__` and `forward`

In [29]:

```
class ReluNet(nn.Module):
    def __init__(self):
        super().__init__()
        ###### TODO QUESTION 2 ########
        self.fc1 =
        self.fc2 =
        self.fc3 =
        ###### TODO QUESTION 2 ########

    def forward(self, x):
        ###### TODO QUESTION 2 ########

        ###### TODO QUESTION 2 ########
        return x
```

In [30]:

```
#Next we'll create an instance of our ReluNet class
relu_net = ReluNet()
```

In [31]:

```
# Here we define the learning rate and the number of epochs for our training.
learning_rate=0.05 #
epochs = 1000  ## usually has already converged around that number

## Here we pick the cost function and the optimizer for our training
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(relu_net.parameters(), lr=learning_rate, momentum=0.9)
```

Training ReluNet

In the next cell the actual training happens. The network is trained for many epochs and for each of the epochs we split our training data into mini batches using the DataLoader that we created previously and iteratively feed them into our network. After each iteration/optimizer step, the parameters are updated in such a way that the loss is minimized.

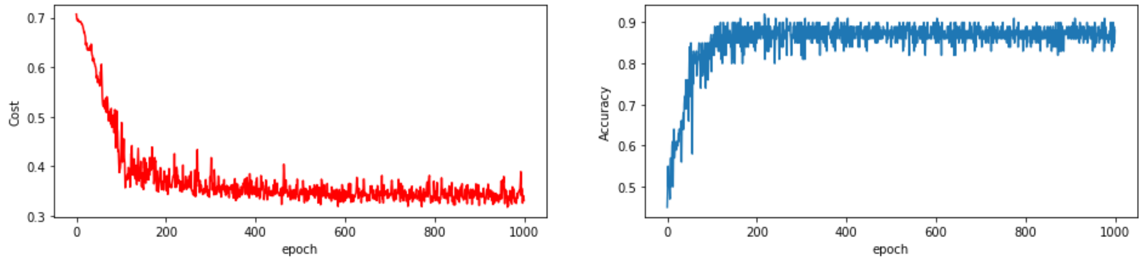
In [32]:

```
start = time.time()

relu_costs=[]
relu_accuracies = []

for epoch in range(epochs):  # loop over the dataset multiple times

    running_loss = 0.0
    num_of_batches = 0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data
#         print(inputs.shape)
#         print(labels.shape)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        logits = relu_net(inputs)
        loss = criterion(logits, labels.type(torch.LongTensor))

        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        num_of_batches +=1

    # calculate test accuracy
    test_logits = relu_net(test_features_tensor)
    _, test_predictions = torch.max(test_logits.data, 1)
    correct_predictions = (test_predictions.int() == test_labels_tensor.int()).s
um().numpy()
    test_length = test_labels_tensor.size()[0]
    accuracy = correct_predictions/test_length
    if epoch%(epochs//20) == 0:
        print('Epoch: %d/%d,  loss: %.3f, test accuracy: %.3f'%(epoch + 1,epochs
, running_loss / num_of_batches, accuracy))
    relu_costs.append(running_loss/num_of_batches)
    relu_accuracies.append(accuracy)


end = time.time()
duration = end-start
print('Finished Training in %d seconds'%(end-start))
```

```
Epoch: 1/1000,  loss: 0.707, test accuracy: 0.450
Epoch: 51/1000,  loss: 0.573, test accuracy: 0.840
Epoch: 101/1000,  loss: 0.447, test accuracy: 0.850
Epoch: 151/1000,  loss: 0.353, test accuracy: 0.830
Epoch: 201/1000,  loss: 0.367, test accuracy: 0.870
Epoch: 251/1000,  loss: 0.367, test accuracy: 0.810
Epoch: 301/1000,  loss: 0.386, test accuracy: 0.900
Epoch: 351/1000,  loss: 0.363, test accuracy: 0.890
Epoch: 401/1000,  loss: 0.351, test accuracy: 0.840
Epoch: 451/1000,  loss: 0.347, test accuracy: 0.900
Epoch: 501/1000,  loss: 0.351, test accuracy: 0.840
Epoch: 551/1000,  loss: 0.339, test accuracy: 0.880
Epoch: 601/1000,  loss: 0.331, test accuracy: 0.890
Epoch: 651/1000,  loss: 0.343, test accuracy: 0.880
Epoch: 701/1000,  loss: 0.343, test accuracy: 0.870
Epoch: 751/1000,  loss: 0.328, test accuracy: 0.870
Epoch: 801/1000,  loss: 0.337, test accuracy: 0.880
Epoch: 851/1000,  loss: 0.335, test accuracy: 0.850
Epoch: 901/1000,  loss: 0.343, test accuracy: 0.880
Epoch: 951/1000,  loss: 0.335, test accuracy: 0.830
Finished Training in 23 seconds
```

In [33]:

```
plot_cost_accuracy(relu_costs, relu_accuracies)
```




```
best_accuracy: 0.92 achieved at epoch: 217
```

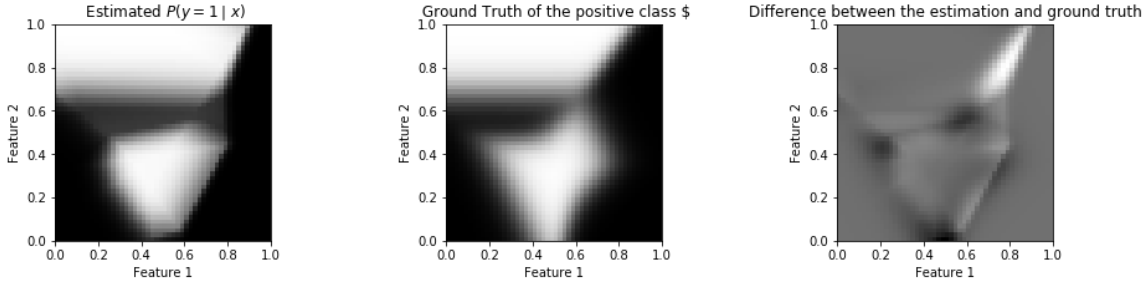
Out[33]:

```
0.92
```

Visualize the posterior of the network trained above

In [34]:

```
visualize_posterior(posterior,relu_net)
```



Cross Validation to pick number of nodes for the hidden layers and the weight decay parameter λ

In this part we are going to use cross validation to try and find the best values for the number of nodes of the hidden layers as well as for the weight decay. We'll use again relu activations.

- **Question 3:** Add the necessary lines inside `__init__` and `forward` in order to define a network that can have a variable number of nodes N. The number of nodes for each layer will be [2, N, N, 2]. Notice that this time the constructor function `__init()__` has a extra argument N, that will allow us to specify the number of nodes when creating an object of this class.

In [35]:

```
class ReluNetVariableNodes(nn.Module):
    def __init__(self,N):
        super().__init__()
        ###### TODO QUESTION 3 ########
        self.N =
        self.fc1 =
        self.fc2 =
        self.fc3 =
        ###### TODO QUESTION 3 ########

    def forward(self, x):
        ###### TODO QUESTION 3 ########

        ###### TODO QUESTION 3 ########
        return x
```

The following cell is only conveniently wrapping the whole training procedure inside a function in order to better organize our code. The function returns the best accuracy achieved.

In [37]:

```
def train_network(training_features, training_labels, test_features, test_labels
, lamda, num_of_nodes, epochs = 1000, learning_rate =0.01):
    start = time.time()
    net = ReluNetVariableNodes(num_of_nodes)

    training_features_tensor = torch.tensor(training_features, dtype=torch.float
)
    training_labels_tensor = torch.tensor(training_labels, dtype=torch.float)
    testing_features_tensor = torch.tensor(test_features, dtype=torch.float, req
uires_grad=False)
    testing_labels_tensor = torch.tensor(test_labels, dtype=torch.float, require
s_grad=False)

    trainset = torch.utils.data.TensorDataset(training_features_tensor, training
_labels_tensor)
    trainloader = torch.utils.data.DataLoader(trainset, batch_size=40, shuffle=T
rue, num_workers=2)

    optimizer = optim.SGD(net.parameters(), lr=learning_rate, weight_decay=lamda
, momentum=0.9)

    costs = []
    accuracies = []

    for epoch in range(epochs):  # loop over the dataset multiple times

        running_loss = 0.0
        num_of_batches = 0
        for i, data in enumerate(trainloader, 0):
            # get the inputs
            inputs, labels = data

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            logits = net(inputs)
            loss = criterion(logits, labels.type(torch.LongTensor))

            loss.backward()
            optimizer.step()

            # print statistics
            running_loss += loss.item()
            num_of_batches +=1

        # calculate test accuracy
        test_logits = net(testing_features_tensor)
        _, test_predictions = torch.max(test_logits.data, 1)
        correct_predictions = (test_predictions.int() == testing_labels_tensor.i
nt()).sum().numpy()
        test_length = testing_labels_tensor.size()[0]
        accuracy = correct_predictions/test_length
        if epoch%(epochs//10) == 0:
            print('Epoch: %d/%d,  loss: %.3f, test accuracy: %.3f'%(epoch + 1,ep
ochs, running_loss / num_of_batches, accuracy))
        costs.append(running_loss/num_of_batches)
        accuracies.append(accuracy)
```

```
    end = time.time()
    duration = end-start
    print('Finished Training in %d seconds'%(end-start))
    best_accuracy = plot_cost_accuracy(costs, accuracies)

    return best_accuracy, net
```

In [38]:

```
def logsample(start, end, num):
    return np.logspace(np.log10(start), np.log10(end), num, base=10.0)

num_of_lamdas = 3
lamda_range = logsample(1e-5, 1e-2, num_of_lamdas)
nodes_range = [5, 10, 20]
```

Compute the cross-validation accuracy for each parameter combination

Now we are going to again use cross-validation (see assignment 3) in order to find the optimal hyperparameters for our network. However instead of splitting our training set into K-Folds, we'll create a single training/validation split which we'll use for our hyperparameter grid search and therefore the *KFold* class from scikit-learn that we used in assignment 3 won't be necessary here.

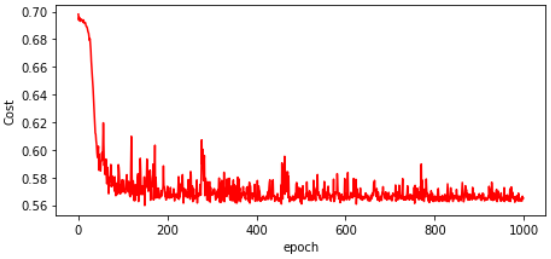
Be warned that this part might take about 10 minutes to run. Try it out for less epochs (maybe 100) to check if everything is working and only then run it for more epochs to improve your results.

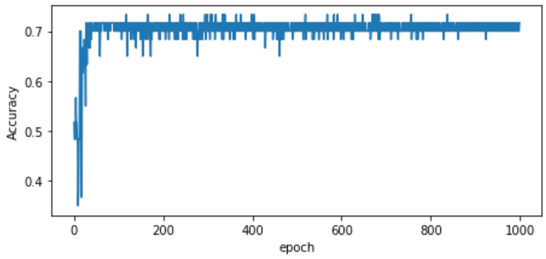
- **Question 4:** Fill in the necessary lines in the double loop iterating over the weight decay and number of nodes (parameter grid-search) in order to perform cross-validation. For each combination of hyperparameters, retrain the network calling the function `train_network()` with the right arguments.

In [39]:

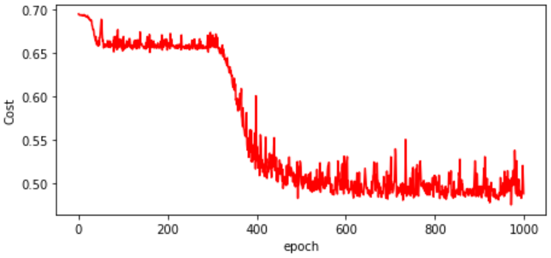
```
cv_start = time.time()

split_percentage = 0.2
length_training = len(training_labels)
validation_split = np.int(np.ceil(split_percentage*length_training))
random_indices = np.random.permutation(np.arange(length_training))
cv_training_set_features = training_features[random_indices[validation_split:]]
cv_training_set_targets = training_labels[random_indices[validation_split:]]
cv_validation_set_features = training_features[random_indices[:validation_split
]]
cv_validation_set_targets = training_labels[random_indices[:validation_split]]
cv_accuracy = np.zeros((len(nodes_range), num_of_lamdas))  # error matrix

learning_rate = 0.05
epochs = 1000

###### TODO QUESTION 4 ########

## loop hyperparameter num_of_nodes
##    loop hyperparameter lamda (i.e. weight decay)

        print('\nCROSS VALIDATION for num_of_nodes = %d and lamda = %f '%(num_of
_nodes, lamda))

        ## add the right arguments for function train_network
        best_accuracy,_ = train_network( )

        # Save the best cv_accuracy for the current combination of parameters
        cv_accuracy[i, j] = best_accuracy

###### TODO QUESTION 4 ########

cv_end = time.time()
print('Finished Grid Search in %d seconds'%(cv_end-cv_start))
```

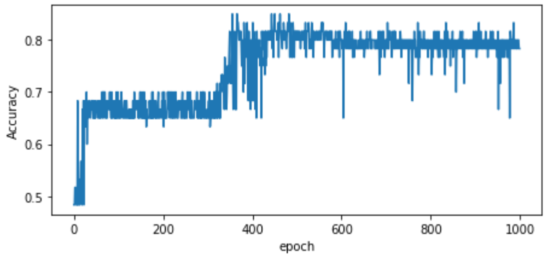
```
CROSS VALIDATION for num_of_nodes = 5 and lamda = 0.000010
Epoch: 1/1000,  loss: 0.698, test accuracy: 0.517
Epoch: 101/1000,  loss: 0.571, test accuracy: 0.700
Epoch: 201/1000,  loss: 0.574, test accuracy: 0.700
Epoch: 301/1000,  loss: 0.567, test accuracy: 0.700
Epoch: 401/1000,  loss: 0.563, test accuracy: 0.700
Epoch: 501/1000,  loss: 0.569, test accuracy: 0.700
Epoch: 601/1000,  loss: 0.568, test accuracy: 0.700
Epoch: 701/1000,  loss: 0.565, test accuracy: 0.717
Epoch: 801/1000,  loss: 0.567, test accuracy: 0.700
Epoch: 901/1000,  loss: 0.568, test accuracy: 0.700
Finished Training in 31 seconds
```



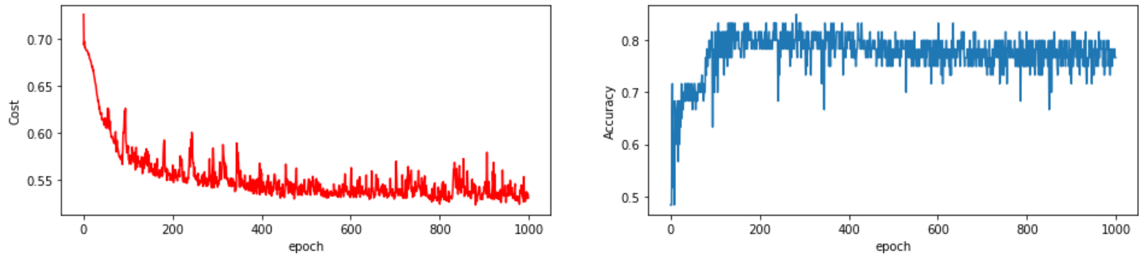



```
best_accuracy: 0.7333333333333333 achieved at epoch: 117

CROSS VALIDATION for num_of_nodes = 5 and lamda = 0.000316
Epoch: 1/1000,  loss: 0.695, test accuracy: 0.483
Epoch: 101/1000,  loss: 0.658, test accuracy: 0.650
Epoch: 201/1000,  loss: 0.658, test accuracy: 0.683
Epoch: 301/1000,  loss: 0.662, test accuracy: 0.667
Epoch: 401/1000,  loss: 0.526, test accuracy: 0.783
Epoch: 501/1000,  loss: 0.501, test accuracy: 0.800
Epoch: 601/1000,  loss: 0.501, test accuracy: 0.800
Epoch: 701/1000,  loss: 0.509, test accuracy: 0.783
Epoch: 801/1000,  loss: 0.499, test accuracy: 0.800
Epoch: 901/1000,  loss: 0.497, test accuracy: 0.783
Finished Training in 19 seconds
```

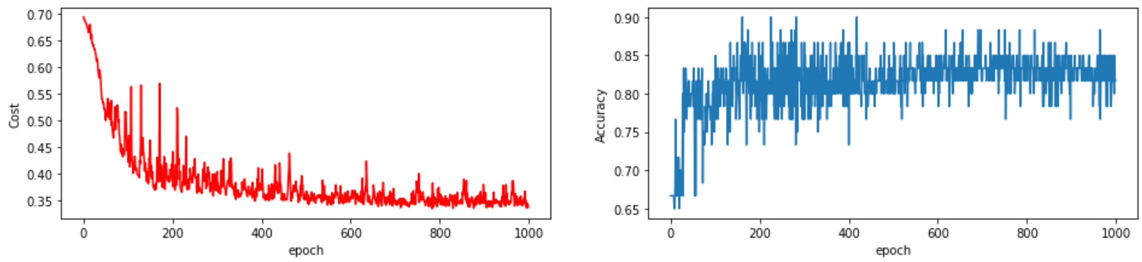




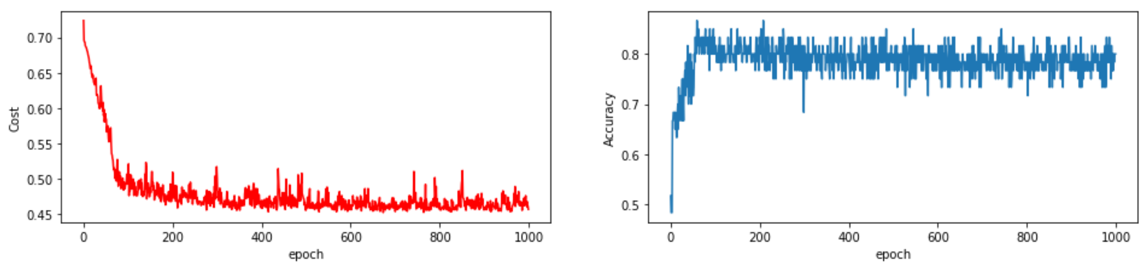

```
best_accuracy: 0.85 achieved at epoch: 355

CROSS VALIDATION for num_of_nodes = 5 and lamda = 0.010000
Epoch: 1/1000,  loss: 0.726, test accuracy: 0.483
Epoch: 101/1000,  loss: 0.586, test accuracy: 0.817
Epoch: 201/1000,  loss: 0.565, test accuracy: 0.783
Epoch: 301/1000,  loss: 0.544, test accuracy: 0.800
Epoch: 401/1000,  loss: 0.557, test accuracy: 0.800
Epoch: 501/1000,  loss: 0.536, test accuracy: 0.783
Epoch: 601/1000,  loss: 0.550, test accuracy: 0.750
Epoch: 701/1000,  loss: 0.532, test accuracy: 0.783
Epoch: 801/1000,  loss: 0.524, test accuracy: 0.750
Epoch: 901/1000,  loss: 0.543, test accuracy: 0.717
Finished Training in 20 seconds
```



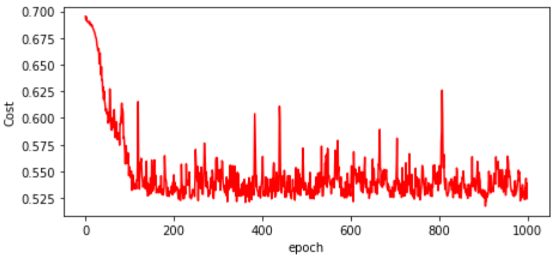

```
best_accuracy: 0.85 achieved at epoch: 282

CROSS VALIDATION for num_of_nodes = 10 and lamda = 0.000010
Epoch: 1/1000,  loss: 0.693, test accuracy: 0.667
Epoch: 101/1000,  loss: 0.422, test accuracy: 0.800
Epoch: 201/1000,  loss: 0.442, test accuracy: 0.783
Epoch: 301/1000,  loss: 0.404, test accuracy: 0.867
Epoch: 401/1000,  loss: 0.387, test accuracy: 0.733
Epoch: 501/1000,  loss: 0.367, test accuracy: 0.800
Epoch: 601/1000,  loss: 0.353, test accuracy: 0.833
Epoch: 701/1000,  loss: 0.343, test accuracy: 0.817
Epoch: 801/1000,  loss: 0.361, test accuracy: 0.833
Epoch: 901/1000,  loss: 0.345, test accuracy: 0.817
Finished Training in 20 seconds
```

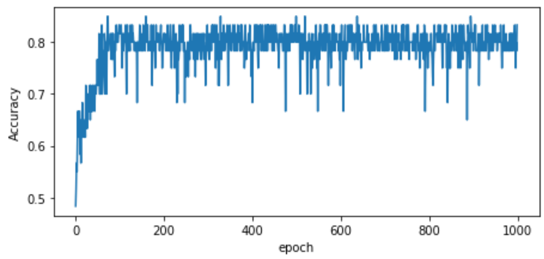



```
best_accuracy: 0.9 achieved at epoch: 161

CROSS VALIDATION for num_of_nodes = 10 and lamda = 0.000316
Epoch: 1/1000,  loss: 0.724, test accuracy: 0.517
Epoch: 101/1000,  loss: 0.505, test accuracy: 0.800
Epoch: 201/1000,  loss: 0.509, test accuracy: 0.817
Epoch: 301/1000,  loss: 0.501, test accuracy: 0.817
Epoch: 401/1000,  loss: 0.469, test accuracy: 0.800
Epoch: 501/1000,  loss: 0.457, test accuracy: 0.783
Epoch: 601/1000,  loss: 0.463, test accuracy: 0.783
Epoch: 701/1000,  loss: 0.454, test accuracy: 0.783
Epoch: 801/1000,  loss: 0.465, test accuracy: 0.783
Epoch: 901/1000,  loss: 0.460, test accuracy: 0.783
Finished Training in 20 seconds
```

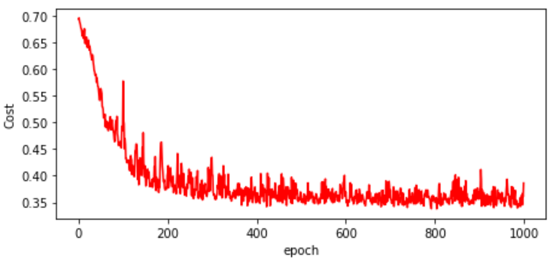


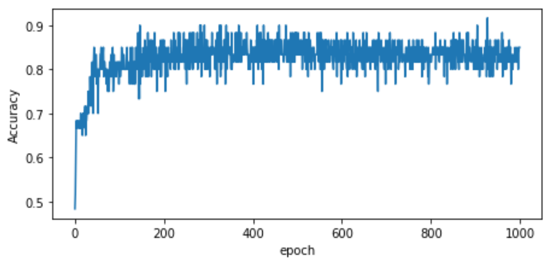
```
best_accuracy: 0.8666666666666667 achieved at epoch: 59

CROSS VALIDATION for num_of_nodes = 10 and lamda = 0.010000
Epoch: 1/1000,  loss: 0.695, test accuracy: 0.483
Epoch: 101/1000,  loss: 0.551, test accuracy: 0.817
Epoch: 201/1000,  loss: 0.547, test accuracy: 0.817
Epoch: 301/1000,  loss: 0.547, test accuracy: 0.817
Epoch: 401/1000,  loss: 0.550, test accuracy: 0.683
Epoch: 501/1000,  loss: 0.528, test accuracy: 0.800
Epoch: 601/1000,  loss: 0.527, test accuracy: 0.800
Epoch: 701/1000,  loss: 0.532, test accuracy: 0.833
Epoch: 801/1000,  loss: 0.547, test accuracy: 0.800
Epoch: 901/1000,  loss: 0.533, test accuracy: 0.783
Finished Training in 20 seconds
```



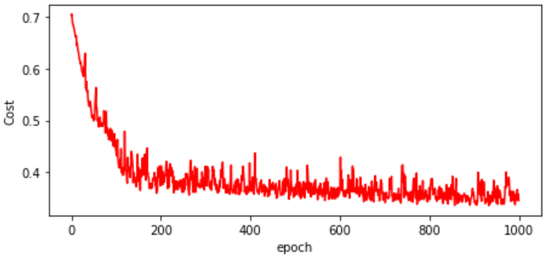


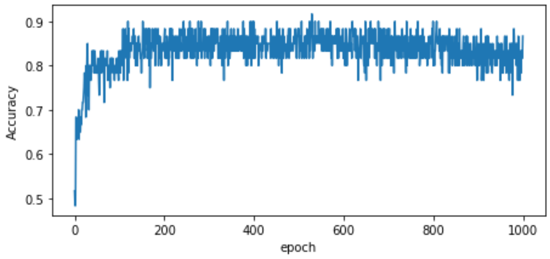

```
best_accuracy: 0.85 achieved at epoch: 72

CROSS VALIDATION for num_of_nodes = 20 and lamda = 0.000010
Epoch: 1/1000,  loss: 0.693, test accuracy: 0.483
Epoch: 101/1000,  loss: 0.577, test accuracy: 0.817
Epoch: 201/1000,  loss: 0.406, test accuracy: 0.867
Epoch: 301/1000,  loss: 0.402, test accuracy: 0.833
Epoch: 401/1000,  loss: 0.353, test accuracy: 0.850
Epoch: 501/1000,  loss: 0.351, test accuracy: 0.833
Epoch: 601/1000,  loss: 0.370, test accuracy: 0.850
Epoch: 701/1000,  loss: 0.373, test accuracy: 0.850
Epoch: 801/1000,  loss: 0.350, test accuracy: 0.817
Epoch: 901/1000,  loss: 0.360, test accuracy: 0.833
Finished Training in 20 seconds
```



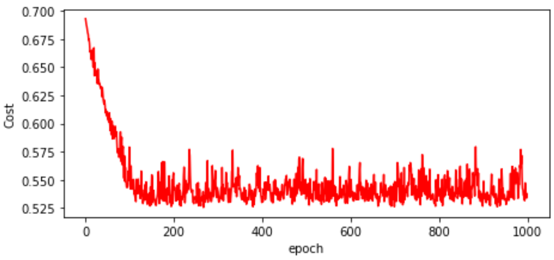


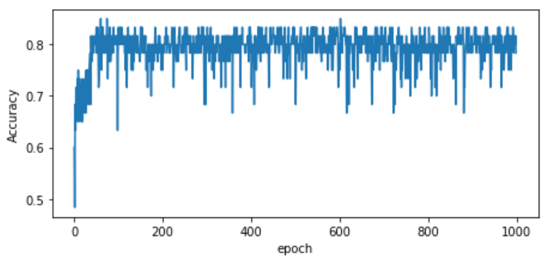

```
best_accuracy: 0.9166666666666666 achieved at epoch: 927

CROSS VALIDATION for num_of_nodes = 20 and lamda = 0.000316
Epoch: 1/1000,  loss: 0.703, test accuracy: 0.517
Epoch: 101/1000,  loss: 0.439, test accuracy: 0.833
Epoch: 201/1000,  loss: 0.411, test accuracy: 0.867
Epoch: 301/1000,  loss: 0.401, test accuracy: 0.867
Epoch: 401/1000,  loss: 0.363, test accuracy: 0.817
Epoch: 501/1000,  loss: 0.356, test accuracy: 0.883
Epoch: 601/1000,  loss: 0.372, test accuracy: 0.850
Epoch: 701/1000,  loss: 0.361, test accuracy: 0.850
Epoch: 801/1000,  loss: 0.355, test accuracy: 0.800
Epoch: 901/1000,  loss: 0.342, test accuracy: 0.850
Finished Training in 20 seconds
```






```
best_accuracy: 0.9166666666666666 achieved at epoch: 529

CROSS VALIDATION for num_of_nodes = 20 and lamda = 0.010000
Epoch: 1/1000,  loss: 0.693, test accuracy: 0.600
Epoch: 101/1000,  loss: 0.566, test accuracy: 0.833
Epoch: 201/1000,  loss: 0.541, test accuracy: 0.750
Epoch: 301/1000,  loss: 0.539, test accuracy: 0.783
Epoch: 401/1000,  loss: 0.535, test accuracy: 0.817
Epoch: 501/1000,  loss: 0.541, test accuracy: 0.783
Epoch: 601/1000,  loss: 0.539, test accuracy: 0.750
Epoch: 701/1000,  loss: 0.549, test accuracy: 0.800
Epoch: 801/1000,  loss: 0.537, test accuracy: 0.783
Epoch: 901/1000,  loss: 0.537, test accuracy: 0.783
Finished Training in 20 seconds
```



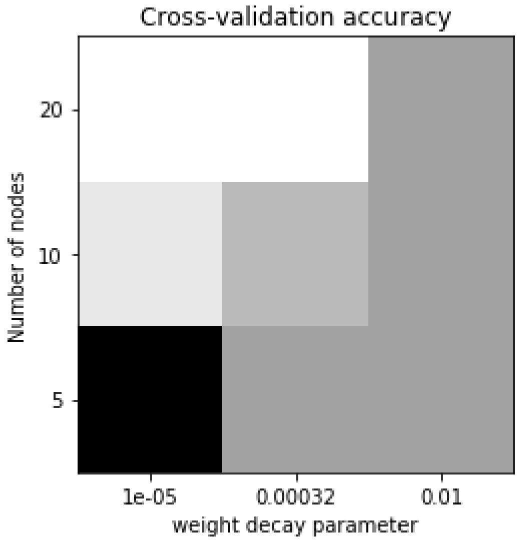



```
best_accuracy: 0.85 achieved at epoch: 60
Finished Grid Search in 195 seconds
```

In [40]:

```
# Display accuracies
fig = plt.figure(figsize=plt.figaspect(0.25))

ax = fig.add_subplot(1, 1, 1)
ax.set_title("Cross-validation accuracy")
ax.set_xlabel("weight decay parameter")
ax.set_ylabel("Number of nodes")
#plt.xticks
plt.imshow(cv_accuracy, cmap='gray', origin='lower')
plt.yticks(range(len(nodes_range)),nodes_range)
plt.xticks(range(len(lamda_range)),np.round(lamda_range,5))
print(cv_accuracy)
```

```
[[0.73333333 0.85       0.85      ]
 [0.9        0.86666667 0.85      ]
 [0.91666667 0.91666667 0.85      ]]
```



In [41]:

```
# Find weight decay  and number of nodes giving the highest accuracy
max_ind = cv_accuracy.argmax()  # index of the max in the flattened array
node_ind, lamda_ind = np.unravel_index(max_ind, cv_accuracy.shape)  # matrix ind
ices
best_lamda = lamda_range[lamda_ind]
best_node_num = nodes_range[node_ind]

print( 'Highest accuracy given by lamda = %f and num_of_nodes=%d'%(best_lamda, b
est_node_num))
```

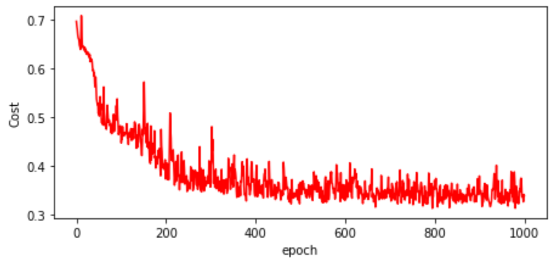
```
Highest accuracy given by lamda = 0.000010 and num_of_nodes=20
```

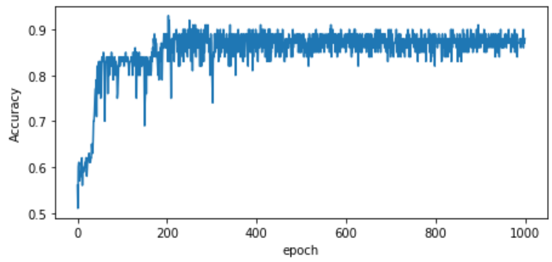
Finally train a network on the full training set using the best combination of parameters

In [42]:

```
# epochs = 1000
best_accuracy, final_net = train_network(training_features, training_labels,
                                            test_features, test_labels,
                                            best_lamda, best_node_num, learning
_rate = learning_rate, epochs = epochs)
```

```
Epoch: 1/1000,  loss: 0.698, test accuracy: 0.560
Epoch: 101/1000,  loss: 0.447, test accuracy: 0.830
Epoch: 201/1000,  loss: 0.387, test accuracy: 0.860
Epoch: 301/1000,  loss: 0.402, test accuracy: 0.830
Epoch: 401/1000,  loss: 0.338, test accuracy: 0.850
Epoch: 501/1000,  loss: 0.366, test accuracy: 0.850
Epoch: 601/1000,  loss: 0.346, test accuracy: 0.880
Epoch: 701/1000,  loss: 0.335, test accuracy: 0.900
Epoch: 801/1000,  loss: 0.354, test accuracy: 0.880
Epoch: 901/1000,  loss: 0.344, test accuracy: 0.860
Finished Training in 22 seconds
```






```
best_accuracy: 0.93 achieved at epoch: 203
```