Exercises for the course
Machine Learning for Data Science
Winter Semester 2022/23

G. Montavon
Institute of Computer Science
Department of Mathematics and Computer Science
Freie Universität Berlin

Exercise Sheet 11 (programming part)

Exercise 3 (25 P)

In this exercise, our objective is to implement a simple neural network from scratch, in particular, error backpropagation and the gradient descent optimization procedure. We first import some useful libraries.

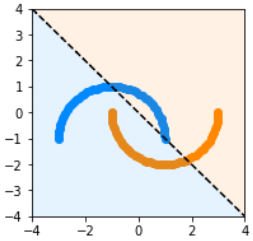
In [135]:
```
import numpy
import matplotlib
%matplotlib inline
from matplotlib import pyplot as plt
na = numpy.newaxis
numpy.random.seed(0)
```

We consider a two-dimensional moon dataset on which to train the network. We also create a grid dataset which we will use to visualize the decision function in two dimensions. We denote our two inputs as x_1 and x_2 and use the suffix `d` and `g` to designate the actual dataset and the grid dataset.

In [141]:
```
# Create a moon dataset on which to train the neural network
import sklearn,sklearn.datasets
Xd,Td = sklearn.datasets.make_moons(n_samples=100)
Xd = Xd*2-1
Td = Td * 2 - 1
X1d = Xd[:,0]
X2d = Xd[:,1]

# Creates a grid dataset on which to inspect the decision function
l = numpy.linspace(-4,4,100)
X1g,X2g = numpy.meshgrid(l,l)
```

The moon dataset is plotted below along with some dummy decision function $x_1 + x_2 = 0$.

In [142]:
```
def plot(Yg,title=None):
    plt.figure(figsize=(3,3))
    plt.scatter(*Xd[Td==-1].T,color='#0088FF')
    plt.scatter(*Xd[Td==1].T,color='#FF8800')
    plt.contour(X1g,X2g,Yg,levels=[0],colors='black',linestyles='dashed')
    plt.contourf(X1g,X2g,Yg,levels=[-100,0,100],colors=['#0088FF','#FF8800'],alpha=0.1)
    if title is not None: plt.title(title)
    plt.show()

plot(X1g+X2g) # plot the dummy decision function
```



We would like to implement the neural network with the equations:

$$\forall_{j=1}^{50} \; : \; z_j = x_1 w_{1j} + x_2 w_{2j} + b_j$$
$$\forall_{j=1}^{50} \; : \; a_j = \max(0, z_j)$$
$$y = \sum_{j=1}^{50} a_j v_j$$

where x_1, x_2 are the two input variables and y is the output of the network. The following code initializes the parameters of the network and implements the forward pass defined above. The neural network is composed of 50 neurons.

In [188]:
```
import numpy

NH = 50

W = numpy.random.normal(0,1/2.0**.5,[2,NH])
B = numpy.random.normal(0,1,[NH])
V = numpy.random.normal(0,1/NH**.5,[NH])

def forward(X1,X2):
    X = numpy.array([X1.flatten(),X2.flatten()]).T # Convert meshgrid into dataset
    Z = X.dot(W)+B
    A = numpy.maximum(0,Z)
    Y = A.dot(V)
    return Y.reshape(X1.shape) # Reshape output into meshgrid
```

We now consider the task of training the neural network to classify the data. For this, we define the error function:

$$\mathcal{E}(\theta) = \frac{1}{N}\sum_{k=1}^{N} \max(0, -y^{(k)}t^{(k)})$$

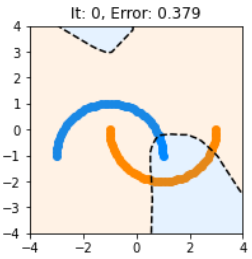
where N is the number of data points, y is the output of the network and t is the label.

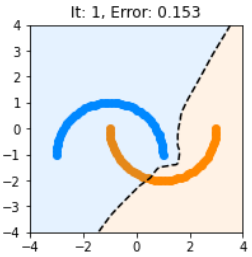
(a) Complete the function below so that it returns the gradient of the error w.r.t. the parameters of the model.

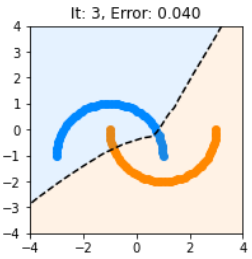
In [189]:
```
def backprop(X1,X2,T):
    X = numpy.array([X1.flatten(),X2.flatten()]).T

    # Compute activations
    Z = X.dot(W)+B
    A = numpy.maximum(0,Z)
    Y = A.dot(V)

    # Compute backward pass
    DY = (-Y*T>0)*(-T)
    DZ = numpy.outer(DY,V)*(Z>0)

    # Compute parameter gradients (averaged over the whole dataset)
    DW = numpy.vstack(((DZ * X[:,0].reshape((-1,1))).mean(axis=0), (DZ * X[:,1].reshape((-1,1))).mean(axis=0)))
    DB = DZ.mean(axis=0)
    DV = (A * DY.reshape((-1,1))).mean(axis=0)


    return DW,DB,DV
```

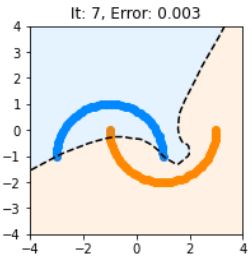
The code below optimizes the network for 32 iterations and at some chosen iterations plots the decision function along with the current error.

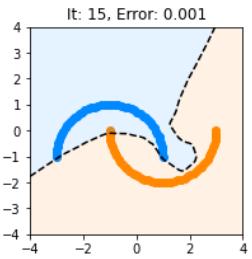
In [190]:
```
for i in range(32):

    if i in [0,1,3,7,15,31]:
        Yg = forward(X1g,X2g)
        Yd = forward(X1d,X2d)
        Ed = numpy.maximum(0,-Yd*Td).mean()
        plot(Yg,title="It: %d, Error: %.3f"%(i,Ed))

    DW,DB,DV = backprop(X1d,X2d,Td)

    W = W - 0.1*DW
    B = B - 0.1*DB
    V = V - 0.1*DV
```

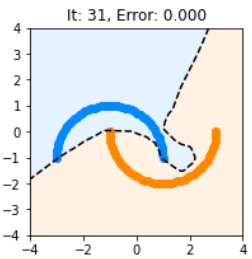












Exercise 4 (25 P)

In this exercise, we would like to train a convolutional neural network on the MNIST dataset. For this, we will make use of the PyTorch libary, which comes with automatic differentiation and predefined neural network layers.

In [191]:
```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms

import utils
```

The following code loads the MNIST data

In [192]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5,), (0.5,))])

trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                      download=True, transform=transform)

testset = torchvision.datasets.MNIST(root='./data', train=False,
                                     download=True, transform=transform)

Xr,Tr = trainset.data.float().view(-1,1,28,28)/127.5-1,trainset.targets
Xt,Tt = testset.data.float().view(-1,1,28,28)/127.5-1,testset.targets
```

We consider for this dataset a convolution network made of four convolutions and two pooling layers.

In [193]:
```
torch.manual_seed(0)
cnn = utils.NNClassifier(nn.Sequential(
    nn.Conv2d( 1, 8, 5), nn.ReLU(), nn.MaxPool2d(2),
    nn.Conv2d( 8, 24, 5), nn.ReLU(), nn.MaxPool2d(2),
    nn.Conv2d( 24, 72, 4), nn.ReLU(),
    nn.Conv2d( 72, 10, 1)
))
```

The network is wrapped in the class `utils.NNClassifier`, which exposes scikit-learn-like functions such as `fit()` and `predict()`. To evaluate the convolutional neural network, we also consider two simpler baselines: a one-layer linear network, and standard fully-connected network composed of two layers.

In [194]:
```
torch.manual_seed(0)
lin = utils.NNClassifier(nn.Sequential(nn.Linear(784, 10)),flat=True)

torch.manual_seed(0)
fc = utils.NNClassifier(nn.Sequential(
    nn.Linear( 784, 512), nn.ReLU(), nn.Linear( 512, 10)
),flat=True)
```

We now proceed with the comparision of these three classifiers.

(a) Train each classifier for 5 epochs and print the classification accuracy on the training and test data (i.e. the fraction of the examples that are correctly classified). To avoid running out of memory, predict the training and test accuracy only based on the 2500 first examples of the training and test set respectively.

In [220]:
```
for name,cl in [('linear',lin),('full',fc),('conv',cnn)]:

    cl.fit(Xr, Tr)
    Tr_predict = numpy.argmax(cl.predict(Xr[:2500]), axis=1)
    Tt_predict = numpy.argmax(cl.predict(Xt[:2500]), axis=1)
    errtr = (numpy.asarray(Tr_predict) == numpy.asarray(Tr[:2500])).sum() / 2500
    errtt = (numpy.asarray(Tt_predict) == numpy.asarray(Tt[:2500])).sum() / 2500

    print('%10s train: %.3f  test: %.3f'%(name,errtr,errtt))
```

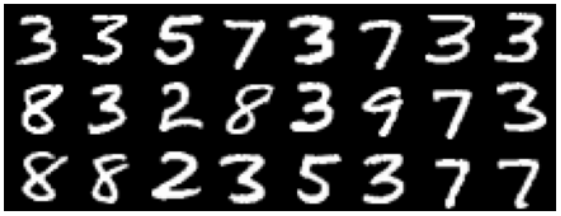
```
linear train: 0.920  test: 0.887
  full train: 0.915  test: 0.894
  conv train: 0.972  test: 0.964
```

We now ask whether some digits are easier to predict than others for the convolutional neural network. For this, we observe that the neural network produces at its output scores y_c for each class c. These scores can be converted to a class probability using the *softargmax* (also called *softmax*) function:

$$p_c = \frac{\exp(y_c)}{\sum_{c'=1}^{10} \exp(y_{c'})}$$

(b) Find for the convolutional network the data points in the test set that are predicted with the highest probability (the lowest being random guessing). To avoid numerical unstability, your implementation should work in the log-probability domain and make use of numerically stable functions of numpy/scipy such as logsumexp.

In [243]:
```
from scipy.special import logsumexp

def softmax(arr):
    return (arr / logsumexp(arr))

Tt_predict = numpy.array(cnn.predict(Xt))
Tt_predict = numpy.apply_along_axis(softmax, 0, Tt_predict).max(axis=1)

highest = Xt[numpy.argpartition(Tt_predict, -24)[-24:]]
lowest = Xt[numpy.argpartition(-Tt_predict, -24)[-24:]]

for digits in [highest,lowest]:
    plt.figure(figsize=(8,3))
    plt.axis('off')
    plt.imshow(digits.numpy().reshape(3,8,28,28).transpose(0,2,1,3).reshape(28*3,28*8),cmap='gray')
    plt.show()
```



We observe that the most confident digits are thick and prototypical. Interestingly, the highest confidence digits are all from the class "3". The low-confidence digits are on the other hand thiner, and are often also more difficult to predict for a human.

In [ ]: