Lab 08: Deep Learning Part I: Fully Connected Neural Networks

In class, we have developed the mathematics and programming techniques for binary classification using fully connected neural networks having one or more hidden layers.

Today, we'll expand on that to consider (small) image classification using again fully connected neural networks with a multinomial (softmax) output layer.

In [1]:
```
!pip install ipywidgets
!jupyter nbextension enable --py widgetsnbextension
```

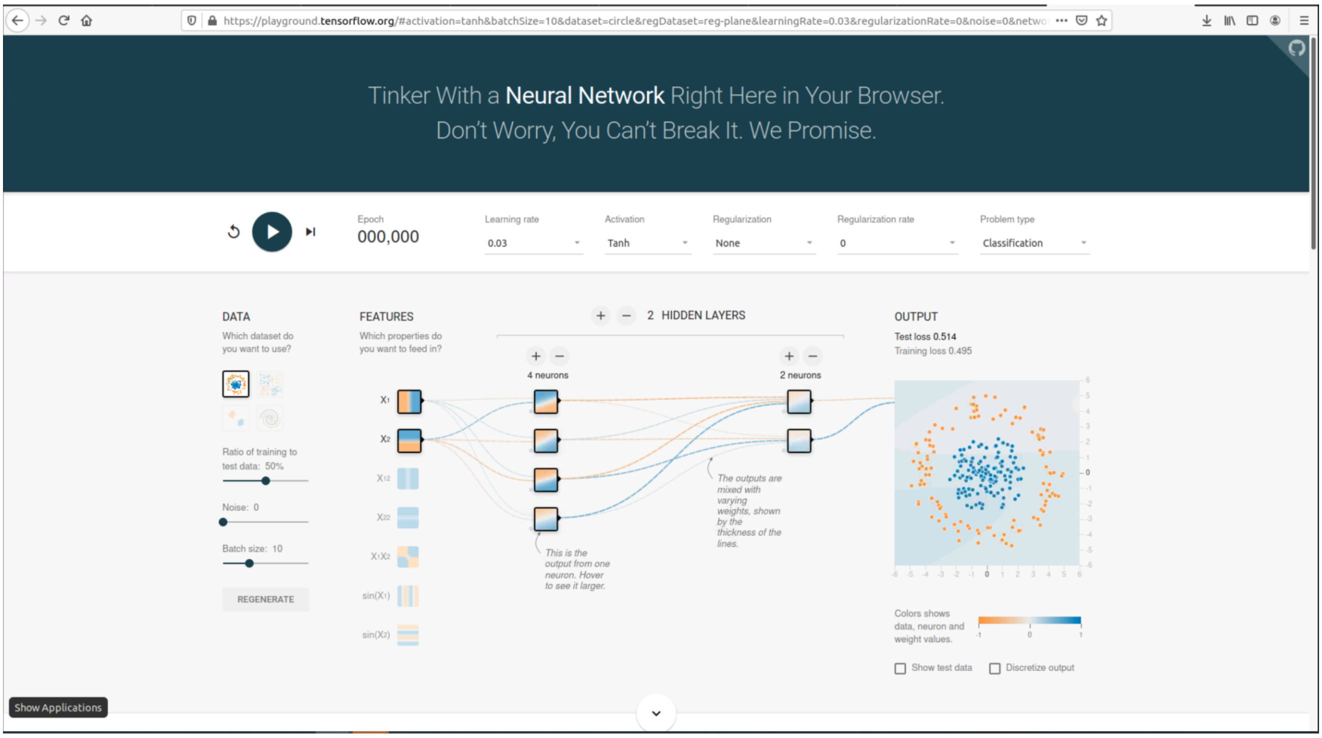
```
Requirement already satisfied: ipywidgets in /home/alisa/anaconda
3/lib/python3.8/site-packages (7.5.1)
Requirement already satisfied: ipython>=4.0.0; python_version >=
"3.3" in /home/alisa/.local/lib/python3.8/site-packages (from ipy
widgets) (7.19.0)
Requirement already satisfied: widgetsnbextension~=3.5.0 in /home
/alisa/anaconda3/lib/python3.8/site-packages (from ipywidgets)
(3.5.1)
Requirement already satisfied: nbformat>=4.2.0 in /home/alisa/ana
conda3/lib/python3.8/site-packages (from ipywidgets) (5.0.7)
Requirement already satisfied: traitlets>=4.3.1 in /home/alisa/an
aconda3/lib/python3.8/site-packages (from ipywidgets) (4.3.3)
Requirement already satisfied: ipykernel>=4.5.1 in /home/alisa/.l
ocal/lib/python3.8/site-packages (from ipywidgets) (5.4.3)
Requirement already satisfied: jedi>=0.10 in /home/alisa/.local/l
ib/python3.8/site-packages (from ipython>=4.0.0; python_version >
= "3.3"->ipywidgets) (0.18.0)
Requirement already satisfied: setuptools>=18.5 in /home/alisa/an
aconda3/lib/python3.8/site-packages (from ipython>=4.0.0; python_
version >= "3.3"->ipywidgets) (49.2.0.post20200714)
Requirement already satisfied: pickleshare in /home/alisa/.local/
lib/python3.8/site-packages (from ipython>=4.0.0; python_version
>= "3.3"->ipywidgets) (0.7.5)
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.
1.0,>=2.0.0 in /home/alisa/.local/lib/python3.8/site-packages (fr
om ipython>=4.0.0; python_version >= "3.3"->ipywidgets) (3.0.14)
Requirement already satisfied: pygments in /home/alisa/.local/lib
/python3.8/site-packages (from ipython>=4.0.0; python_version >=
"3.3"->ipywidgets) (2.7.4)
Requirement already satisfied: backcall in /home/alisa/.local/lib
/python3.8/site-packages (from ipython>=4.0.0; python_version >=
"3.3"->ipywidgets) (0.2.0)
Requirement already satisfied: decorator in /home/alisa/.local/li
b/python3.8/site-packages (from ipython>=4.0.0; python_version >=
"3.3"->ipywidgets) (4.4.2)
Requirement already satisfied: pexpect>4.3; sys_platform != "win3
2" in /home/alisa/anaconda3/lib/python3.8/site-packages (from ipy
thon>=4.0.0; python_version >= "3.3"->ipywidgets) (4.8.0)
Requirement already satisfied: notebook>=4.4.1 in /home/alisa/ana
conda3/lib/python3.8/site-packages (from widgetsnbextension~=3.5.
0->ipywidgets) (6.0.3)
Requirement already satisfied: jupyter-core in /home/alisa/.local
/lib/python3.8/site-packages (from nbformat>=4.2.0->ipywidgets)
(4.7.0)
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /home/a
lisa/.local/lib/python3.8/site-packages (from nbformat>=4.2.0->ip
ywidgets) (3.2.0)
Requirement already satisfied: ipython-genutils in /home/alisa/.l
ocal/lib/python3.8/site-packages (from nbformat>=4.2.0->ipywidget
s) (0.2.0)
Requirement already satisfied: six in /home/alisa/anaconda3/lib/p
ython3.8/site-packages (from traitlets>=4.3.1->ipywidgets) (1.15.
0)
Requirement already satisfied: jupyter-client in /home/alisa/.loc
al/lib/python3.8/site-packages (from ipykernel>=4.5.1->ipywidget
s) (6.1.11)
Requirement already satisfied: tornado>=4.2 in /home/alisa/anacon
da3/lib/python3.8/site-packages (from ipykernel>=4.5.1->ipywidget
```

```
s) (6.0.4)
Requirement already satisfied: parso<0.9.0,>=0.8.0 in /home/alisa
/.local/lib/python3.8/site-packages (from jedi>=0.10->ipython>=4.
0.0; python_version >= "3.3"->ipywidgets) (0.8.1)
Requirement already satisfied: wcwidth in /home/alisa/.local/lib/
python3.8/site-packages (from prompt-toolkit!=3.0.0,!=3.0.1,<3.1.
0,>=2.0.0->ipython>=4.0.0; python_version >= "3.3"->ipywidgets)
(0.2.5)
Requirement already satisfied: ptyprocess>=0.5 in /home/alisa/ana
conda3/lib/python3.8/site-packages (from pexpect>4.3; sys_platfor
m != "win32"->ipython>=4.0.0; python_version >= "3.3"->ipywidget
s) (0.6.0)
Requirement already satisfied: pyzmq>=17 in /home/alisa/anaconda3
/lib/python3.8/site-packages (from notebook>=4.4.1->widgetsnbexte
nsion~=3.5.0->ipywidgets) (19.0.1)
Requirement already satisfied: prometheus-client in /home/alisa/.
local/lib/python3.8/site-packages (from notebook>=4.4.1->widgetsn
bextension~=3.5.0->ipywidgets) (0.9.0)
Requirement already satisfied: nbconvert in /home/alisa/anaconda3
/lib/python3.8/site-packages (from notebook>=4.4.1->widgetsnbexte
nsion~=3.5.0->ipywidgets) (5.6.1)
Requirement already satisfied: Send2Trash in /home/alisa/anaconda
3/lib/python3.8/site-packages (from notebook>=4.4.1->widgetsnbext
ension~=3.5.0->ipywidgets) (1.5.0)
Requirement already satisfied: jinja2 in /home/alisa/.local/lib/p
ython3.8/site-packages (from notebook>=4.4.1->widgetsnbextension~
=3.5.0->ipywidgets) (3.0.1)
Requirement already satisfied: terminado>=0.8.1 in /home/alisa/an
aconda3/lib/python3.8/site-packages (from notebook>=4.4.1->widget
snbextension~=3.5.0->ipywidgets) (0.8.3)
Requirement already satisfied: attrs>=17.4.0 in /home/alisa/anaco
nda3/lib/python3.8/site-packages (from jsonschema!=2.5.0,>=2.4->n
bformat>=4.2.0->ipywidgets) (19.3.0)
Requirement already satisfied: pyrsistent>=0.14.0 in /home/alisa
/.local/lib/python3.8/site-packages (from jsonschema!=2.5.0,>=2.4
->nbformat>=4.2.0->ipywidgets) (0.17.3)
Requirement already satisfied: python-dateutil>=2.1 in /home/alis
a/anaconda3/lib/python3.8/site-packages (from jupyter-client->ipy
kernel>=4.5.1->ipywidgets) (2.8.1)
Requirement already satisfied: mistune<2,>=0.8.1 in /home/alisa/a
naconda3/lib/python3.8/site-packages (from nbconvert->notebook>=
4.4.1->widgetsnbextension~=3.5.0->ipywidgets) (0.8.4)
Requirement already satisfied: pandocfilters>=1.4.1 in /home/alis
a/anaconda3/lib/python3.8/site-packages (from nbconvert->notebook
>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets) (1.4.2)
Requirement already satisfied: testpath in /home/alisa/anaconda3/
lib/python3.8/site-packages (from nbconvert->notebook>=4.4.1->wid
getsnbextension~=3.5.0->ipywidgets) (0.4.4)
Requirement already satisfied: bleach in /home/alisa/anaconda3/li
b/python3.8/site-packages (from nbconvert->notebook>=4.4.1->widge
tsnbextension~=3.5.0->ipywidgets) (3.1.5)
Requirement already satisfied: entrypoints>=0.2.2 in /home/alisa/
anaconda3/lib/python3.8/site-packages (from nbconvert->notebook>=
4.4.1->widgetsnbextension~=3.5.0->ipywidgets) (0.3)
Requirement already satisfied: defusedxml in /home/alisa/anaconda
3/lib/python3.8/site-packages (from nbconvert->notebook>=4.4.1->w
idgetsnbextension~=3.5.0->ipywidgets) (0.6.0)
Requirement already satisfied: MarkupSafe>=2.0 in /home/alisa/.lo
```

```
cal/lib/python3.8/site-packages (from jinja2->notebook>=4.4.1->wi
dgetsnbextension~=3.5.0->ipywidgets) (2.0.1)
Requirement already satisfied: packaging in /home/alisa/anaconda3
/lib/python3.8/site-packages (from bleach->nbconvert->notebook>=
4.4.1->widgetsnbextension~=3.5.0->ipywidgets) (20.4)
Requirement already satisfied: webencodings in /home/alisa/anacon
da3/lib/python3.8/site-packages (from bleach->nbconvert->notebook
>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets) (0.5.1)
Requirement already satisfied: pyparsing>=2.0.2 in /home/alisa/.l
ocal/lib/python3.8/site-packages (from packaging->bleach->nbconve
rt->notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets) (2.4.
7)
Enabling notebook extension jupyter-js-widgets/extension...
      - Validating: OK
```

What is Deep learning doing?

Let's try to classify the deep learning in this [link](https://playground.tensorflow.org/). The page can observe your network visualization when learning it.



Select The initial setup of data (at the left) as:

- Ratio of training to test data: 90%
- Noise: 5
- Batch size: 4

Press run and observe the result.

Exercise 1 (10 points)

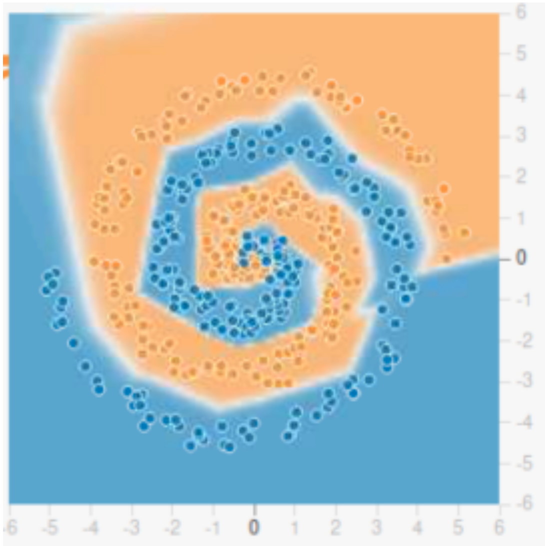
Select the spiral shape (the 4th shape). Select The initial setup of data (at the left) as:

- Ratio of training to test data: 90%
- Noise: 35
- Batch size: 4

Try to make the best separating result. Capture your **FULL** screen and input below

Capture scrren here!

Expect result:



Special coding

You can make jupyter in GUI (it also can export to HTML website).

In [2]:
```
from IPython.display import display, Markdown, clear_output
# widget packages
import ipywidgets as widgets

# defining some widgets
lblQ1 = widgets.Label(
        value="Q1) What is your learning rate?")
rdoQ1 = widgets.RadioButtons(
        options=['0.00001', '0.0001', '0.001', '0.003', '0.01', '
0.03', '0.1', '0.3', '1', '3', '10',],
        value='0.03',
        description='\t',
        disabled=False)

lblQ2 = widgets.Label(
        value="Q2) What is your activation function in last laye
r?")
rdoQ2 = widgets.RadioButtons(
        options=['ReLu', 'Tanh', 'Sigmoid', 'Linear',],
        value='Tanh',
        description='\t',
        disabled=False)

lblQ3 = widgets.Label(
        value="Q3) What is problem type?")
rdoQ3 = widgets.RadioButtons(
        options=['Classification', 'Regression'],
        value='Classification',
        description='\t',
        disabled=False)

lblQ4 = widgets.Label(value="Q4) Which input do you use?")
chkQ4_1 = widgets.Checkbox(
           description='$X_1$',
           value=True)
chkQ4_2 = widgets.Checkbox(
           description='$X_2$',
           value=True)
chkQ4_3 = widgets.Checkbox(
           description='$X_1^2$',)
chkQ4_4 = widgets.Checkbox(
           description='$X_1X_2$',)
chkQ4_5 = widgets.Checkbox(
           description='$X_2^2$',)
chkQ4_6 = widgets.Checkbox(
           description='sin$(X_1)$',)
chkQ4_7 = widgets.Checkbox(
           description='sin$(X_2)$',)
chkQ4 = widgets.VBox([chkQ4_1, chkQ4_2, chkQ4_3, chkQ4_4, chkQ4_
5, chkQ4_6, chkQ4_7])

lblQ5 = widgets.Label(value="Q5) How many hidden layers do you us
e?")
txtQ5 = widgets.IntText(
       value=0,
       description='hidden layers', )
```

```
lblQ6 = widgets.Label(value="Q6) Explain your nodes for each laye
r")
txtQ6 = widgets.Textarea(
       value='',
       description='Explain here', )

box = widgets.VBox([lblQ1, rdoQ1, lblQ2, rdoQ2, lblQ3, rdoQ3, lbl
Q4, chkQ4, lblQ5, txtQ5, lblQ6, txtQ6,])

box
```

In [3]:

```
q4str = ""
if chkQ4_1.value:
    q4str += " X1,"
if chkQ4_2.value:
    q4str += " X2,"
if chkQ4_3.value:
    q4str += " X1^2,"
if chkQ4_4.value:
    q4str += " X1X2,"
if chkQ4_5.value:
    q4str += " X2^2,"
if chkQ4_6.value:
    q4str += " sin(X1),"
if chkQ4_7.value:
    q4str += " sin(X2),"
print("Use input features:", q4str)
print("Problem type:", rdoQ3.value)
print("The last activation function:", rdoQ2.value)
print("Learning rate:", rdoQ1.value)
print("Use", txtQ5.value, "hidden layers. Each layer contains", t
xtQ6.value)
```

```
Use input features:  X1, X2,
Problem type: Classification
The last activation function: Tanh
Learning rate: 0.03
Use 0 hidden layers. Each layer contains
```

MNIST Data

An image is a 2D array of pixels. Pixels can be scalar intensities (for a grayscale / black and white image) or a vector indicating a point in a color space such as RGB or HSV.

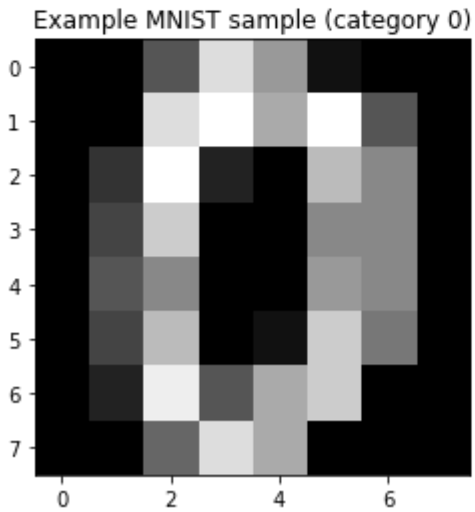
Today we'll consider 8x8 grayscale images of digits from the famous "MNIST" dataset, which was considered a benchmark for machine learning algorithms up to the early 2000s, before the advent of large-scale image classification datasets.

This dataset in SciKit-Learn has 10 classes, with 180 samples per class in most cases, for a total of 1797 samples.

Let's load the dataset and plot an example.

In [4]:
```
import numpy as np
from sklearn.datasets import load_digits
import matplotlib.pyplot as plt

# Load data

data = load_digits()

def convert_to_one_hot(y):
    y_vect = np.zeros((len(y), 10))
    for i in range(len(y)):
        y_vect[i, int(y[i])] = 1
    return y_vect

# Convert target indices to one-hot representation

y_indices = data.target
y = convert_to_one_hot(y_indices)
X = np.matrix(data.data)
M = X.shape[0]
N = X.shape[1]

# Plot an example

plt.imshow(np.reshape(X[0,:],(8,8)), 'gray')
plt.title('Example MNIST sample (category %d)' % y_indices[0])
```

Out[4]:
```
Text(0.5, 1.0, 'Example MNIST sample (category 0)')
```



Hand-Coded Fully Connected Neural Network

OK, now let's modify our code from class to work with this dataset and run 100 epochs of training. The main change is to use a one-hot encoding of the 10 classes at the output layer and to use the softmax activation function at the output. Some minor changes are required to calculate multinomial cross entropy loss rather than binary cross entropy loss.

In [5]:
```
import random
import warnings
warnings.filterwarnings("ignore")
```

Normalize each input feature

In [6]:
```
def normalize(X):
    M = X.shape[0]
    XX = X - np.tile(np.mean(X,0),[M,1])
    XX = np.divide(XX, np.tile(np.std(XX,0),[M,1]))
    return np.nan_to_num(XX, copy=True,nan=0.0)

XX = normalize(X)
```

Partion data into training and testing dataset

In [7]:
```
idx = np.arange(0,M)

random.shuffle(idx)
percent_train = .6
m_train = int(M * percent_train)
train_idx = idx[0:m_train]
test_idx = idx[m_train:M+1]
X_train = XX[train_idx,:];
X_test = XX[test_idx,:];

y_train = y[train_idx];
y_test = y[test_idx];
y_test_indices = y_indices[test_idx]
```

Create network

Let's start with a 3-layer network with sigmoid activation functions, \ 6 units in layer 1, and 5 units in layer 2.

In [8]:
```
h2 = 5
h1 = 6
W = [[], np.random.normal(0,0.1,[N,h1]),
         np.random.normal(0,0.1,[h1,h2]),
         np.random.normal(0,0.1,[h2,10])]
b = [[], np.random.normal(0,0.1,[h1,1]),
         np.random.normal(0,0.1,[h2,1]),
         np.random.normal(0,0.1,[10,1])]
L = len(W)-1
```

Create some important functions

In [9]:
```
def sigmoid_act(z):
    return 1/(1+np.exp(-z))

def softmax_act(z):
    exps = np.exp(z)
    return exps / np.sum(exps)

def sigmoid_actder(z):
    az = sigmoid_act(z)
    prod = np.multiply(az,1-az)
    return prod

def ff(x,W,b):
    L = len(W)-1
    a = x
    for l in range(1,L+1):
        z = W[l].T*a+b[l]
        if (l == L):
            a = softmax_act(z)
        else:
            a = sigmoid_act(z)
    return a

def loss(y, yhat):
    return - np.dot(y, np.log(yhat))

def forward(x_this, W, b):
    L = len(W)-1
    a = [x_this]
    z = [[]]
    delta = [[]]
    dW = [[]]
    db = [[]]
    for l in range(1,L+1):
        z.append(W[l].T*a[l-1]+b[l])
        if (l == L):
            a.append(softmax_act(z[l]))
        else:
            a.append(sigmoid_act(z[l]))
        # Just to give arrays the right shape for the backprop st
ep
        delta.append([]); dW.append([]); db.append([])
    return a, z, delta, dW, db

def back_propagation(y_this, a, z, W, dW, db, show_check=False):
    '''
    Backprop step. Note that derivative of multinomial cross entr
opy
    loss is the same as that of binary cross entropy loss. See
    https://levelup.gitconnected.com/killer-combo-softmax-and-cro
ss-entropy-5907442f60ba
    for a nice derivation.
    '''
    L = len(W)-1
    delta[L] = a[L] - np.matrix(y_this).T
    for l in range(L,0,-1):
        db[l] = delta[l].copy()
```

```
        dW[l] = a[l-1] * delta[l].T
        if l > 1:
            delta[l-1] = np.multiply(sigmoid_actder(z[l-1]), W[l]
* delta[l])

    # Check delta calculation

    if show_check:
        print('Target: %f' % y_this)
        print('y_hat: %f' % a[L][0,0])
        print(db)
        y_pred = ff(x_this,W,b)
        diff = 1e-3
        W[1][10,0] = W[1][10,0] + diff
        y_pred_db = ff(x_this,W,b)
        L1 = loss(y_this,y_pred)
        L2 = loss(y_this,y_pred_db)
        db_finite_difference = (L2-L1)/diff
        print('Original out %f, perturbed out %f' %
             (y_pred[0,0], y_pred_db[0,0]))
        print('Theoretical dW %f, calculated db %f' %
              (dW[1][10,0], db_finite_difference[0,0]))
    return dW, db

def update_step(W, b, dW, db, alpha):
    L = len(W)-1
    for l in range(1,L+1):
        W[l] = W[l] - alpha * dW[l]
        b[l] = b[l] - alpha * db[l]
    return W, b
```

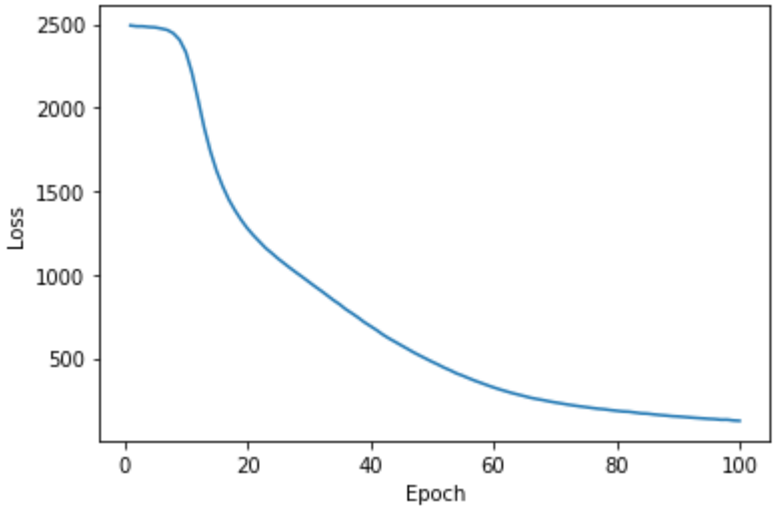
Train for 100 epochs with mini-batch size 1

In [10]:
```
cost_arr = []

alpha = 0.01
max_iter = 100
for iter in range(0, max_iter):
    loss_this_iter = 0
    order = np.random.permutation(m_train)
    for i in range(0, m_train):

        # Grab the pattern order[i]

        x_this = X_train[order[i],:].T
        y_this = y_train[order[i],:]

        # Feed forward step
        a, z, delta, dW, db = forward(x_this, W, b)

        # calulate loss
        loss_this_pattern = loss(y_this, a[L])
        loss_this_iter = loss_this_iter + loss_this_pattern

        # back propagation
        dW, db = back_propagation(y_this, a, z, W, dW, db, show_c
heck=False)

        # update weight, bias
        W, b = update_step(W, b, dW, db, alpha)

    cost_arr.append(loss_this_iter[0,0])
    print('Epoch %d train loss %f' % (iter, loss_this_iter))
```

```
Epoch 0 train loss 2492.344257
Epoch 1 train loss 2487.525511
Epoch 2 train loss 2487.238954
Epoch 3 train loss 2483.200943
Epoch 4 train loss 2481.067671
Epoch 5 train loss 2474.016761
Epoch 6 train loss 2465.600345
Epoch 7 train loss 2445.221863
Epoch 8 train loss 2406.182667
Epoch 9 train loss 2333.825817
Epoch 10 train loss 2209.137173
Epoch 11 train loss 2047.016641
Epoch 12 train loss 1879.336879
Epoch 13 train loss 1738.824428
Epoch 14 train loss 1621.855275
Epoch 15 train loss 1528.842471
Epoch 16 train loss 1449.328902
Epoch 17 train loss 1385.076851
Epoch 18 train loss 1328.267116
Epoch 19 train loss 1278.077211
Epoch 20 train loss 1235.552163
Epoch 21 train loss 1197.409254
Epoch 22 train loss 1159.159291
Epoch 23 train loss 1128.422710
Epoch 24 train loss 1096.571133
Epoch 25 train loss 1067.686544
Epoch 26 train loss 1039.051829
Epoch 27 train loss 1011.680095
Epoch 28 train loss 985.120406
Epoch 29 train loss 957.417071
Epoch 30 train loss 929.993867
Epoch 31 train loss 903.687563
Epoch 32 train loss 875.962890
Epoch 33 train loss 847.773737
Epoch 34 train loss 822.844242
Epoch 35 train loss 793.356435
Epoch 36 train loss 769.295410
Epoch 37 train loss 743.953903
Epoch 38 train loss 716.105615
Epoch 39 train loss 692.582827
Epoch 40 train loss 668.772138
Epoch 41 train loss 642.392926
Epoch 42 train loss 619.346141
Epoch 43 train loss 598.530682
Epoch 44 train loss 577.495193
Epoch 45 train loss 557.009923
Epoch 46 train loss 535.606918
Epoch 47 train loss 516.481480
Epoch 48 train loss 497.986134
Epoch 49 train loss 479.249121
Epoch 50 train loss 462.021279
Epoch 51 train loss 443.300060
Epoch 52 train loss 427.460825
Epoch 53 train loss 409.137051
Epoch 54 train loss 395.518946
Epoch 55 train loss 380.155592
Epoch 56 train loss 365.161938
Epoch 57 train loss 352.089791
```

```
Epoch 58 train loss 338.377155
Epoch 59 train loss 325.581631
Epoch 60 train loss 313.576330
Epoch 61 train loss 302.657262
Epoch 62 train loss 291.885403
Epoch 63 train loss 282.479671
Epoch 64 train loss 273.322458
Epoch 65 train loss 263.618022
Epoch 66 train loss 256.222092
Epoch 67 train loss 249.695236
Epoch 68 train loss 242.211875
Epoch 69 train loss 235.416772
Epoch 70 train loss 229.522407
Epoch 71 train loss 222.961371
Epoch 72 train loss 217.537023
Epoch 73 train loss 212.062268
Epoch 74 train loss 207.460251
Epoch 75 train loss 202.341826
Epoch 76 train loss 197.556330
Epoch 77 train loss 194.391872
Epoch 78 train loss 189.428347
Epoch 79 train loss 185.249477
Epoch 80 train loss 181.467380
Epoch 81 train loss 179.655660
Epoch 82 train loss 174.830572
Epoch 83 train loss 169.906574
Epoch 84 train loss 168.297092
Epoch 85 train loss 163.824275
Epoch 86 train loss 159.951417
Epoch 87 train loss 157.017714
Epoch 88 train loss 153.145123
Epoch 89 train loss 151.081070
Epoch 90 train loss 148.271006
Epoch 91 train loss 145.232111
Epoch 92 train loss 142.357805
Epoch 93 train loss 139.771827
Epoch 94 train loss 137.077505
Epoch 95 train loss 135.341032
Epoch 96 train loss 131.743902
Epoch 97 train loss 132.033622
Epoch 98 train loss 127.268589
Epoch 99 train loss 125.200760
```

In [11]:
```
plt.plot(np.arange(1,max_iter+1,1), cost_arr)
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.show()
```



Get test set accuracy

In [12]:
```
def predict_y(W, b, X):
    M = X.shape[0]
    y_pred = np.zeros(M)
    for i in range(X.shape[0]):
        y_pred[i] = np.argmax(ff(X[i,:].T, W, b))
    return y_pred

y_test_predicted = predict_y(W, b, X_test)
y_correct = y_test_predicted == y_test_indices
test_accuracy = np.sum(y_correct) / len(y_correct)

print('Test accuracy: %.4f' % (test_accuracy))
```

```
Test accuracy: 0.8929
```

In-class exercise (40 points)

Modify the code above to plot both training loss and test loss as a function of epoch number. Use early stopping to obtain the best model according to the validation set. Experiment with the hyperparameters (learning rate, number of layers, number of units per layer) to get the best result you can.

- Do at least 3 examples
- Plot graphs
- Tell the validation accuracy
- Describe your experiments and results in your lab report.

In [ ]:
```
# modify your code here
```

PyTorch tutorial

Is there an easier way to build this type of model? One way is to learn a framework such as TensorFlow or PyTorch. Both of these frameworks have their pros and cons, but PyTorch is probably the most productive neural network framework for research purposes. We'll use it here.

The material for this tutorial is from [Anand Saha's PyTorch tutorial (https://github.com/anandsaha/deep.learning.with.pytorch)](https://github.com/anandsaha/deep.learning.with.pytorch).

Tensors and Tensor operations

Let's get some hands on experience with tensor creation and operations. The torch package contains the necessary data structures to create multidimensional tensors. It also defines the mathematical operations that can be performed on these.

In [13]:
```
import torch
```

In [14]:
```
print(torch.__version__)
```

```
1.7.1
```

Tensor creation

Create a (2x3) dimentional Tensor.

Note that a) You get back a FloatTensor b) The values are uninitialized

In [15]:
```
t = torch.Tensor(2, 3)
print(t)
```

```
tensor([[-8.8324e-20,  4.5715e-41, -8.8324e-20],
        [ 4.5715e-41,  3.2155e-18,  4.5713e-41]])
```

The above call was equivalent to

In [16]:
```
t = torch.FloatTensor(2, 3)
print(t)
```

```
tensor([[1.0410e-31, 3.0840e-41, 2.1163e-37],
        [2.1234e-37, 1.9324e-37, 1.7228e-34]])
```

Inspect type of an element

In [17]:
```
t[0][0]
```

Out[17]:
```
tensor(1.0410e-31)
```

In [18]:
```
type(t[0][0])
```

Out[18]:
```
torch.Tensor
```

Inspect `t` 's dimensions

In [19]:
```
print(t.size())
print(t.dim())
print(len(t.size()) == t.dim())
```

```
torch.Size([2, 3])
2
True
```

Set values

In [20]:
```
t[0][0] = 1
t[0][1] = 2
t[0][2] = 3
t[1][0] = 4
t[1][1] = 5
t[1][2] = 6
print(t)
```

```
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

Let's cast a FloatTensor to IntTensor

In [21]:
```
t = torch.FloatTensor([1.1, 2.2])
print(t)
t.type(torch.IntTensor)
```

```
tensor([1.1000, 2.2000])
```

Out[21]:
```
tensor([1, 2], dtype=torch.int32)
```

Let's explore some other ways of creating a tensor

In [22]:
```
# From another Tensor

t2 = torch.Tensor(t)
print(t2)
```

```
tensor([1.1000, 2.2000])
```

In [23]:
```
# From a Python list

t3 = torch.IntTensor([[1, 2],[3, 4]])
print(t3)
```

```
tensor([[1, 2],
        [3, 4]], dtype=torch.int32)
```

In [24]:
```
# From a NumPy array

import numpy as np
a = np.array([55, 66])
t4 = torch.Tensor(a)
print(t4)
```

```
tensor([55., 66.])
```

In [25]:
```
# Create a Tensor with all zeros

t5 = torch.zeros(2, 3)
print(t5)
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

In [26]:
```
# Create a Tensor with all ones

t6 = torch.ones(2, 3)
print(t6)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

In [27]:
```
# Create a Tensor with all ones with dimensions
# of another Tensor

t7 = torch.ones_like(t4)
print(t7)
```

```
tensor([1., 1.])
```

Tensor operations

Add two Tensors

In [28]:
```
t1 = torch.ones(2, 2)
t2 = torch.ones(2, 2)
t = t1 + t2
print(t)
```

```
tensor([[2., 2.],
        [2., 2.]])
```

Inplace/out-of-place operations

In [29]:
```
t1.add(t2)
```
Out[29]:
```
tensor([[2., 2.],
        [2., 2.]])
```
In [30]:
```
print(t1)
```
```
tensor([[1., 1.],
        [1., 1.]])
```
In [31]:
```
t1.add_(t2)
```
Out[31]:
```
tensor([[2., 2.],
        [2., 2.]])
```
In [32]:
```
print(t1)
```
```
tensor([[2., 2.],
        [2., 2.]])
```

Class methods and package functions

In [33]:
```
t1.cos()
```
Out[33]:
```
tensor([[-0.4161, -0.4161],
        [-0.4161, -0.4161]])
```
In [34]:
```
torch.cos(t1)
```
Out[34]:
```
tensor([[-0.4161, -0.4161],
        [-0.4161, -0.4161]])
```

A few more operations

In [35]:
```
# Create a one-dimensional tensor of steps equally
# spaced points between start and end

torch.linspace(3, 10, steps=5)
```
Out[35]:
```
tensor([ 3.0000,  4.7500,  6.5000,  8.2500, 10.0000])
```
In [36]:
```
# Create a 1-D Tensor with values from [start, end)

torch.arange(0, 5)
```
Out[36]:
```
tensor([0, 1, 2, 3, 4])
```

In [37]:
```
# Create a (2x3) Tensor with random values sampled
# from uniform distrubution on the interval [0,1)

torch.rand((2,3))
```

Out[37]:
```
tensor([[0.3599, 0.4225, 0.0156],
        [0.0617, 0.4219, 0.4685]])
```

In [38]:
```
# Create a (2x3) Tensor with random values sampled
# from normal distrubution with 0 mean and variance 1

torch.randn((2,3))
```

Out[38]:
```
tensor([[-0.5529, -0.8400, -0.4847],
        [ 0.2756,  0.0774, -0.1100]])
```

In [39]:
```
# Do a matrix multiply

a = torch.rand((2, 3))
b = torch.rand((3, 2))

torch.mm(a, b)
```

Out[39]:
```
tensor([[0.2847, 0.8870],
        [0.0251, 0.2303]])
```

Variables

Next, let's understand variables in PyTorch and the operations we can perform on them.

In [40]:
```
import torch
from torch.autograd import Variable
```

Let's create a small computation graph

In [41]:
```
x = Variable(torch.FloatTensor([11.2]), requires_grad=True)
y = 2 * x
```

In [42]:
```
print(x)
print(y)
```

```
tensor([11.2000], requires_grad=True)
tensor([22.4000], grad_fn=<MulBackward0>)
```

In [43]:
```
print(x.data)
print(y.data)
```

```
tensor([11.2000])
tensor([22.4000])
```

In [44]:
```
print(x.grad_fn)
print(y.grad_fn)
```

```
None
<MulBackward0 object at 0x7f6e226cdb80>
```

In [45]:
```
y.backward() # Calculates the gradients
```

In [46]:
```
print(x.grad)
print(y.grad)
```

```
tensor([2.])
None
```

Working with PyTorch and NumPy

In [47]:
```
import torch
import numpy as np
```

Convert a NumPy array to Tensor

In [48]:
```
n = np.array([2, 3])
t = torch.from_numpy(n)
print(n)
print(t)
```

```
[2 3]
tensor([2, 3])
```

Change a Tensor value, and see the change in corresponding NumPy array

In [49]:
```
n[0] = 100
print(t)
```

```
tensor([100,   3])
```

Convert a Tensor to NumPy array

In [50]:
```
t = torch.FloatTensor([5, 6])
n = t.numpy()
print(t)
print(n)
```

```
tensor([5., 6.])
[5. 6.]
```

Change a Tensor value, and see the change in corresponding NumPy array

In [51]:
```
t[0] = 100
print(n)
```

```
[100.   6.]
```

Tensors on GPU

Check if your machine has GPU support

In [52]:
```
if torch.cuda.is_available():
    print("GPU Supported")
else:
    print("GPU Not Supported")
```

```
GPU Supported
```

Check the number of GPUs attached to this machine

In [53]:
```
torch.cuda.device_count()
```

Out[53]:
```
2
```

Get device name

In [54]:
```
torch.cuda.get_device_name(0)
```

Out[54]:
```
'GeForce RTX 2080 Ti'
```

Moving a Tensor to GPU

In [55]:
```
t = torch.FloatTensor([2, 3])
```

In [56]:
```
print(t)
```

```
tensor([2., 3.])
```

In [57]:
```
t = t.cuda(0)
```

Creating a Tensor on GPU, directly

In [58]:
```
t = torch.cuda.FloatTensor([2, 3])
print(t)
```

```
tensor([2., 3.], device='cuda:0')
```

Bring it back to CPU

In [59]:
```
t = t.cpu()
print(t)
```

```
tensor([2., 3.])
```

Use device context

In [60]:
```
with torch.cuda.device(0):
    t = torch.cuda.FloatTensor([2, 3])
    print(t)
```

```
tensor([2., 3.], device='cuda:0')
```

In [ ]:
```

```

MNIST digit recognition using PyTorch

This part of the lab was taken from the [Kaggle tutorial on MNIST with PyTorch ('https://www.kaggle.com/justuser/mnist-with-pytorch-fully-connected-network)](https://www.kaggle.com/justuser/mnist-with-pytorch-fully-connected-network).

We will use a fully connected neural network and a batch learning algorithm and explain each step along the way.

So, with that being said, let's start with imports that we will need. First of all, we need to import PyTorch. There are some common names for torch modules (like numpy is always named np): torch.nn.functional is imported as F, torch.nn is the core module, and is simply imported as nn. Also, we need numpy. We also use pyplot and seaborn for visualization, but they are not required for the network itself. And finally, we use pandas for importing and transforming data.

In [61]:
```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")
```

Now we can import and transform the data. I decided to split it into input and labels right away at this step:

In [63]:
```
print("Reading the data...")
data = pd.read_csv('train_mnist.csv', sep=",")
test_data = pd.read_csv('test_mnist.csv', sep=",")

print("Reshaping the data...")
dataFinal = data.drop('label', axis=1)
labels = data['label']


dataNp = dataFinal.to_numpy()
labelsNp = labels.to_numpy()
test_dataNp = test_data.to_numpy()

print("Data is ready")
```

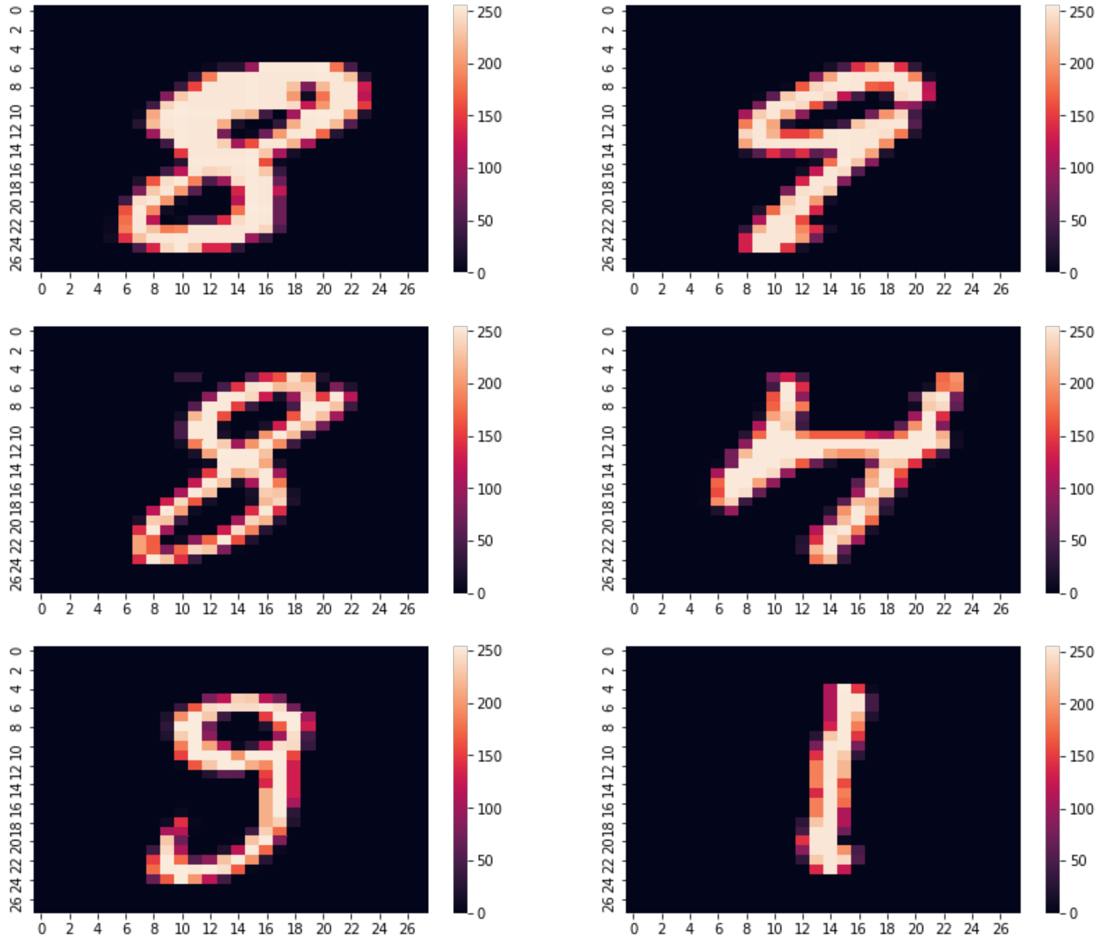
```
Reading the data...
Reshaping the data...
Data is ready
```

Now that data is ready, we can take a look at what we're dealing with. I will be using heatmaps from seaborn, which is an excellent tool for matrix visualization. But first, since the images in the MNIST dataset are represented as a long 1d arrays of pixels, we will need to reshape it into 2d array. That's where .reshape() from numpy comes in handy. The pictures are 28 x 28 pixels, so these will be the parameters.

Let's select a couple random samples and visualize them. I will also print their labels, so we can compare images with their actual value:

In [64]:
```
plt.figure(figsize=(14, 12))

pixels = dataNp[10].reshape(28, 28)
plt.subplot(321)
sns.heatmap(data=pixels)

pixels = dataNp[11].reshape(28, 28)
plt.subplot(322)
sns.heatmap(data=pixels)

pixels = dataNp[20].reshape(28, 28)
plt.subplot(323)
sns.heatmap(data=pixels)

pixels = dataNp[32].reshape(28, 28)
plt.subplot(324)
sns.heatmap(data=pixels)

pixels = dataNp[40].reshape(28, 28)
plt.subplot(325)
sns.heatmap(data=pixels)

pixels = dataNp[52].reshape(28, 28)
plt.subplot(326)
sns.heatmap(data=pixels)

print(labels[10], " / ", labels[11])
print(labels[20], " / ", labels[32])
print(labels[40], " / ", labels[52])
```

```
8  /  9
8  /  4
9  /  1
```



PyTorch has it's own way to store data - those are called tensors, and they are just like numpy arrays, but are suited for PyTorch needs. If we want to feed the data to the network, we need to transform the dataset into those tensors. The good news is that PyTorch can easily do that by transforming numpy arrays or regular lists into tensors.

In [65]:
```
x = torch.FloatTensor(dataNp.tolist())
y = torch.LongTensor(labelsNp.tolist())
```

Before we start writing the actual network, we need to determine what will be the hyperparameters. Those will not be adjusted during training, so we need to be careful how we set them up.

Here's what we will specify:

- **input_size** - size of the input layer, it is always fixed (784 pixels)
- **output_size** - size of the output layer, also fixed size (10 for every possible digit)
- **hidden_size** - size of the hidden layer, this parameter determines structure of the network. 200 worked for me, but it is worth to play with this parameter to see what works for you
- **epochs** - how many times will the network go through the entire dataset during training.
- **learning_rate** - determines how fast will the network learn. You should be very careful about this parameter, because if it is too high, the network won't learn at all, if it is too low, the net will learn too long. I's always about balance. Usualy 10^-3 - 10^-5 works just fine.
- **batch_size** - size of mini batches during training

In [66]:
```
# hyperparameters
input_size = 784
output_size = 10
hidden_size = 200

epochs = 20
batch_size = 50
learning_rate = 0.00005
```

Now we can finally write the actual network. To make it all work, the Network class needs to inherit the *nn.Module*, which gives it the basic functionality required, and allows PyTorch to work with it as expected.

When writing a PyTorch neural network, some things must always be there:

- __init__(self) - initializes the net and creates an instance of that *nn.Module*. Here we define the structure of the network.
- forward(self, x) - defines forward propagation and how the data flow through the network. Of course, it is based on the structure that is defined in the previous function.

In the initialization, first of all, we need to initialize super (or base) module that the net inherits. After that first line, is the definition of structure. You can experiment with (put more layers or change hidden layer size, etc.), but this structure worked for me just fine.

In forward propagation we simply reassign the value of x as it flows through the layers and return the [softmax (https://en.wikipedia.org/wiki/Softmax_function)](https://en.wikipedia.org/wiki/Softmax_function) at the end.

In [67]:
```
class Network(nn.Module):

    def __init__(self):
        super(Network, self).__init__()
        self.l1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.l3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.l1(x)
        x = self.relu(x)
        x = self.l3(x)
        return F.log_softmax(x)
```

After we've defined the network, we can initialize it. Also, if we "print" the instance of the net, we can see the structure of it in a neat format:

In [68]:
```
net = Network()
print(net)
```

```
Network(
  (l1): Linear(in_features=784, out_features=200, bias=True)
  (relu): ReLU()
  (l3): Linear(in_features=200, out_features=10, bias=True)
)
```

Now it's time to set up the optimizer (http://pytorch.org/docs/master/optim.html) and a loss function.

There are quite a lot of things happening behind these two lines of code, so if you don't know what is going on here, don't worry too much for now, it will get clearer eventualy.

Optimizer is what updates the parameters of the network. I will be using Stochastic Gradient Descent with momentum. Also, the optimizer takes the network parameters as an argument, but it's not a big deal since we can get those with .parameters() function.

I decided to use Cross Entropy Loss (https://en.wikipedia.org/wiki/Cross_entropy) for this problem, but again, there are many options and you are free to choose whatever suits you best.

In [69]:
```
optimizer = optim.SGD(net.parameters(), lr=learning_rate, momentu
m=0.9)
loss_func = nn.CrossEntropyLoss()
```

Now that everything is ready, our network can start learning. I will separate data into minibatches and feed it to the network. It has many advantages over single batch learning, but that is a different story.

Also, I will use loss_log list to keep track of the loss function during the training process.

In [70]:
```
loss_log = []

for e in range(epochs):
    for i in range(0, x.shape[0], batch_size):
        x_mini = x[i:i + batch_size]
        y_mini = y[i:i + batch_size]

        x_var = Variable(x_mini)
        y_var = Variable(y_mini)

        optimizer.zero_grad()
        net_out = net(x_var)

        loss = loss_func(net_out, y_var)
        loss.backward()
        optimizer.step()

        if i % 100 == 0:
            loss_log.append(loss.item())

    print('Epoch: {} - Loss: {:.6f}'.format(e, loss.item()))
```

```
Epoch: 0 - Loss: 0.069158
Epoch: 1 - Loss: 0.039167
Epoch: 2 - Loss: 0.022598
Epoch: 3 - Loss: 0.014092
Epoch: 4 - Loss: 0.011621
Epoch: 5 - Loss: 0.009669
Epoch: 6 - Loss: 0.005499
Epoch: 7 - Loss: 0.005262
Epoch: 8 - Loss: 0.004254
Epoch: 9 - Loss: 0.003897
Epoch: 10 - Loss: 0.003131
Epoch: 11 - Loss: 0.002975
Epoch: 12 - Loss: 0.002338
Epoch: 13 - Loss: 0.002036
Epoch: 14 - Loss: 0.001913
Epoch: 15 - Loss: 0.001443
Epoch: 16 - Loss: 0.001274
Epoch: 17 - Loss: 0.001236
Epoch: 18 - Loss: 0.000946
Epoch: 19 - Loss: 0.000951
```

So, let's go line by line and see what is happening here:

This is the main loop that goes through all the epochs of training. An epoch is one full training on the full dataset.

```
for e in range(epochs):
```

This is the inner loop that simply goes through the dataset batch by batch:

```
for i in range(0, x.shape[0], batch_size):
```

Here is where we get the batches out of our data and simply assign them to variables for further work:

```
x_mini = x[i:i + batch_size]
y_mini = y[i:i + batch_size]
```

These two lines are quite *important*. Remember I told you about tensors and how PyTorch stores data in them? That's not the end of story. Actually, to allow the network to work with data, we need a wrapper for those tensors called Variable. It has some additional properties, like allowing automatic gradient computation when backpropagating. It is required for the proper work of PyTorch, so we will add them here and supply tensors as parameters:

```
x_var = Variable(x_mini)
y_var = Variable(y_mini)
```

This line just resets the gradient of the optimizer:

```
optimizer.zero_grad()
```

Remember the *forward(self, x)* function that we previously defined? The next line is basically calling this function and does the forward propagation:

```
net_out = net(x_var)
```

This line computes the loss function based on predictions of the net and the correct answers:

```
loss = loss_func(net_out, y_var)
```

Here we compute the gradient based on the loss that we've got. It will be used to adjust parameters of the network.

```
loss.backward()
```

And here is where we finally update our network with new adjusted parameters:

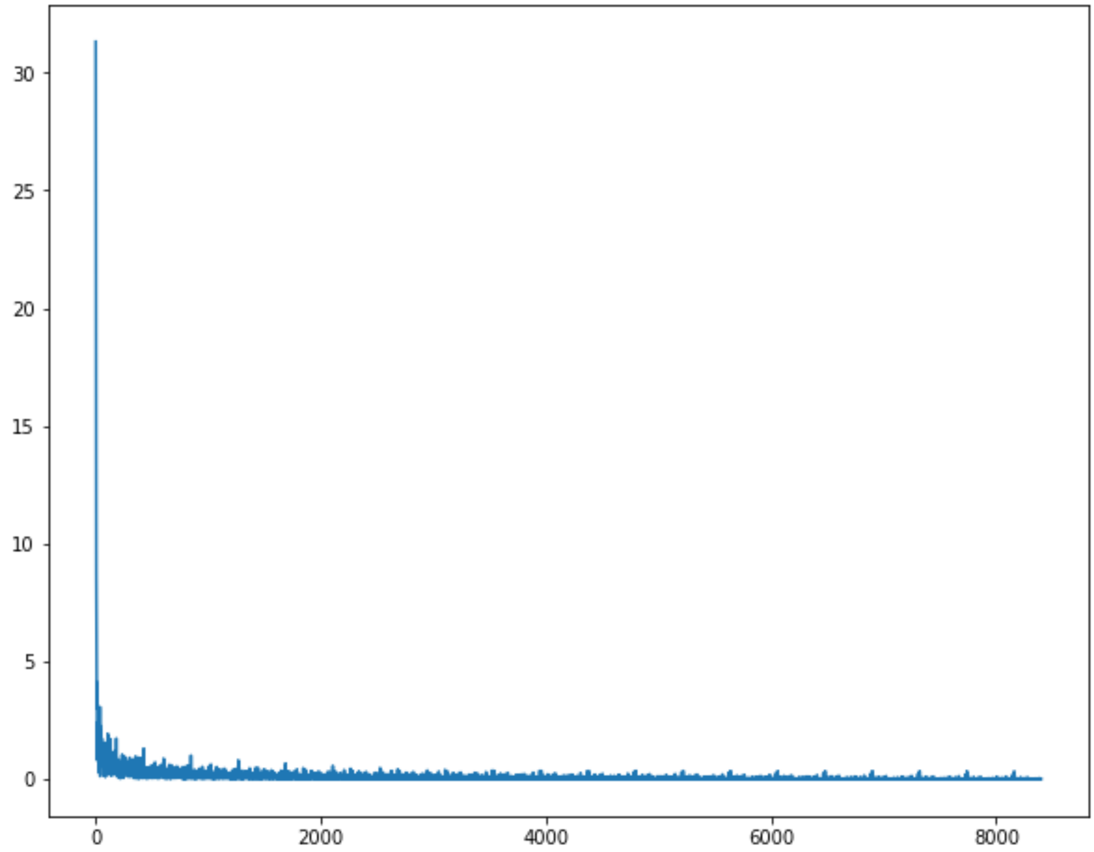
```
optimizer.step()
```

The rest is just logging, which might be helpful to observe how well the network is performing.

After the network is done with training, we can take a look at the loss function, and how it behaved during training:

In [71]:
```
plt.figure(figsize=(10,8))
plt.plot(loss_log)
```

Out[71]:
```
[<matplotlib.lines.Line2D at 0x7f6db3731610>]
```



At this point, the network should be trained, and we can make a prediction using the test dataset. All we need to do is wrap the data into the Variable and feed it to the trained net, so nothing new here.

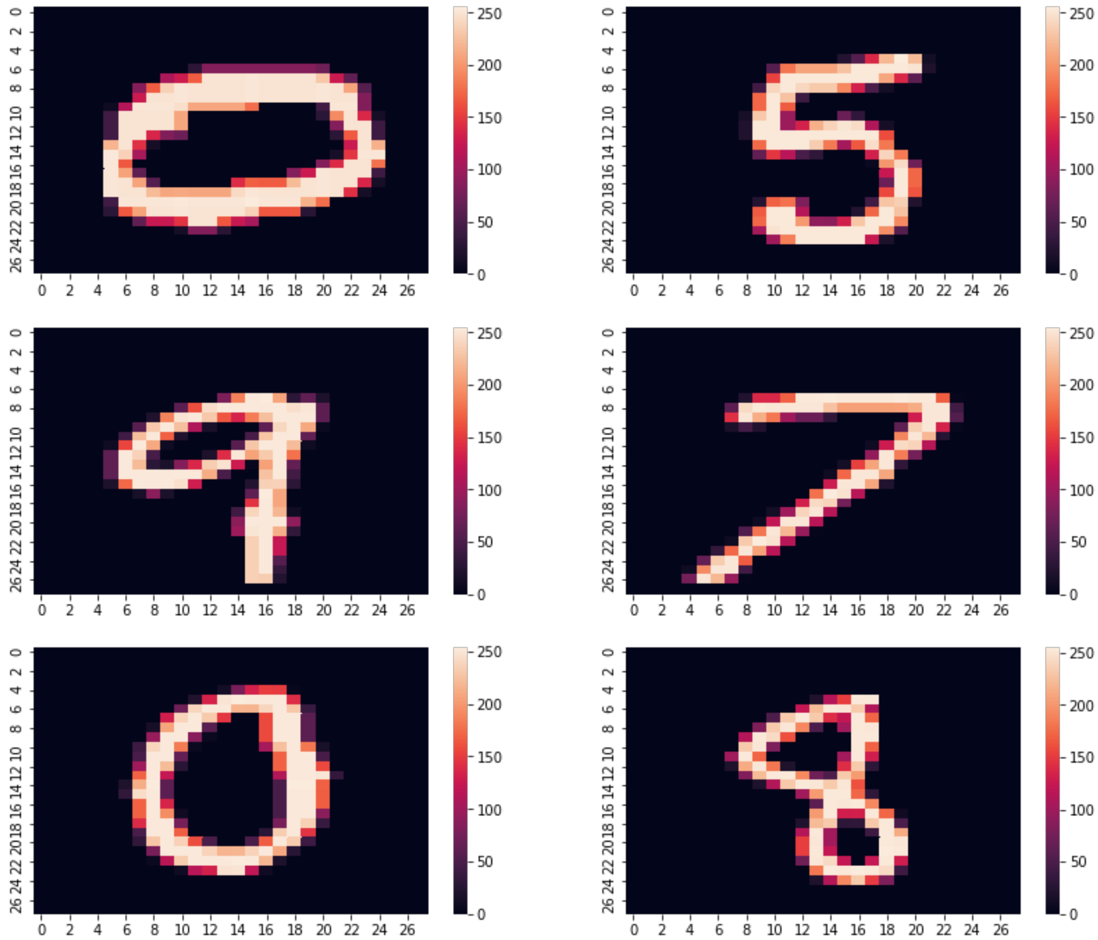
In [72]:
```
test = torch.FloatTensor(test_dataNp.tolist())
test_var = Variable(test)

net_out = net(test_var)

print(torch.max(net_out.data, 1)[1].numpy())
```

```
[2 0 9 ... 3 9 2]
```

Now we have out predictions that are ready to be submitted. Before that, we can take a look at predictions and compare them to the actual pictures of digits, just like at the start with training data:

In [73]:
```
plt.figure(figsize=(14, 12))

pixels = test_dataNp[1].reshape(28, 28)
plt.subplot(321)
sns.heatmap(data=pixels)
test_sample = torch.FloatTensor(test_dataNp[1].tolist())
test_var_sample = Variable(test_sample)
net_out_sample = net(test_var_sample)

pixels = test_dataNp[10].reshape(28, 28)
plt.subplot(322)
sns.heatmap(data=pixels)
test_sample = torch.FloatTensor(test_dataNp[10].tolist())
test_var_sample = Variable(test_sample)
net_out_sample = net(test_var_sample)

pixels = test_dataNp[20].reshape(28, 28)
plt.subplot(323)
sns.heatmap(data=pixels)
test_sample = torch.FloatTensor(test_dataNp[20].tolist())
test_var_sample = Variable(test_sample)
net_out_sample = net(test_var_sample)

pixels = test_dataNp[30].reshape(28, 28)
plt.subplot(324)
sns.heatmap(data=pixels)
test_sample = torch.FloatTensor(test_dataNp[30].tolist())
test_var_sample = Variable(test_sample)
net_out_sample = net(test_var_sample)

pixels = test_dataNp[100].reshape(28, 28)
plt.subplot(325)
sns.heatmap(data=pixels)
test_sample = torch.FloatTensor(test_dataNp[100].tolist())
test_var_sample = Variable(test_sample)
net_out_sample = net(test_var_sample)

pixels = test_dataNp[2000].reshape(28, 28)
plt.subplot(326)
sns.heatmap(data=pixels)
test_sample = torch.FloatTensor(test_dataNp[1].tolist())
test_var_sample = Variable(test_sample)
net_out_sample = net(test_var_sample)


print("Prediction: {} / {}".format(torch.max(net_out.data, 1)[1].
numpy()[1], torch.max(net_out.data, 1)[1].numpy()[10]))
print("Prediction: {} / {}".format(torch.max(net_out.data, 1)[1].
numpy()[20], torch.max(net_out.data, 1)[1].numpy()[30]))
print("Prediction: {} / {}".format(torch.max(net_out.data, 1)[1].
numpy()[100], torch.max(net_out.data, 1)[1].numpy()[2000]))
```

```
Prediction: 0 / 5
Prediction: 9 / 7
Prediction: 0 / 8
```



In [74]:
```
output = (torch.max(net_out.data, 1)[1]).numpy()
#np.savetxt("out.csv", np.dstack((np.arange(1, output.size+1),output))[0],"%d,%d",header="ImageId,Label")
```

And that is about it, we've made a simple neural network using PyTorch that can recognize handwritten digits. Not so bad!

When I was writing this notebook, this model scorred 96.6%, which is not perfect by any means, but it's not that bad either.

I hope this was useful for some of you. If you are totally new to deep learning, I suggest you learn how the neural networks actually work from the inside, especially the backpropagation algorithm.

These videos explain [neural nets (https://www.youtube.com/watch?v=aircAruvnKk&t=708s)](https://www.youtube.com/watch?v=aircAruvnKk&t=708s) and [backpropagation (https://www.youtube.com/watch?v=Ilg3gGewQ5U)](https://www.youtube.com/watch?v=Ilg3gGewQ5U) quite well.

Also I suggest you to take a look at this [online book (http://neuralnetworksanddeeplearning.com/chap1.html)](http://neuralnetworksanddeeplearning.com/chap1.html) (it's absolutely free, btw), where neural networks are explained in great detail, and it even has an implementation of the MNIST problem from scratch, using only numpy.

If you have any feedback, feel free to leave comments down below, and good luck with your deep learning adventures :)

Take-home exercise (50 points)

Make sure you can run the PyTorch examples of MNIST classification, then apply the PyTorch example to another classification problem you've worked with this semester, for example the breast cancer dataset. Get familiar with working with models in PyTorch.

Report your experiments and results in your brief lab report.

In [75]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

data = pd.read_csv("breast_cancer.csv")
print(data.columns)
```

```
Index(['id', 'diagnosis', 'radius_mean', 'texture_mean', 'perimet
er_mean',
       'area_mean', 'smoothness_mean', 'compactness_mean', 'conca
vity_mean',
       'concave points_mean', 'symmetry_mean', 'fractal_dimension
_mean',
       'radius_se', 'texture_se', 'perimeter_se', 'area_se', 'smo
othness_se',
       'compactness_se', 'concavity_se', 'concave points_se', 'sy
mmetry_se',
       'fractal_dimension_se', 'radius_worst', 'texture_worst',
       'perimeter_worst', 'area_worst', 'smoothness_worst',
       'compactness_worst', 'concavity_worst', 'concave points_wo
rst',
       'symmetry_worst', 'fractal_dimension_worst', 'Unnamed: 32
'],
      dtype='object')
```

In [76]:
```
X = data.drop(columns=['id','Unnamed: 32','diagnosis'])
y = data['diagnosis']
y, unique_y = pd.factorize(y)
X_columns = X.columns

X_train, X_test, y_train, y_test = train_test_split(X.values, y,
test_size=0.2, random_state=42)

scaler = StandardScaler()
scaler.fit(X_train)

X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

X_train_tensor = torch.FloatTensor(X_train)
X_test_tensor = torch.FloatTensor(X_test)
y_train_tensor = torch.LongTensor(y_train)
y_test_tensor = torch.LongTensor(y_test)
```

In [ ]:
```
# Your code here

input_size = None
output_size = None
#hidden1_size = None
#hidden2_size = None
#hidden3_size = None
#hidden4_size = None

epochs = None
batch_size = None
learning_rate = None
```

In [ ]:
```
class Network(nn.Module):

    def __init__(self):
        super(Network, self).__init__()
        ### BEGIN SOLUTION
        self.l1 = nn.Linear(input_size, hidden1_size)
        self.relu1 = nn.ReLU()
        self.l3 = nn.Linear(hidden1_size, output_size)
        ### END SOLUTION

    def forward(self, x):
        ### BEGIN SOLUTION
        x = self.l1(x)
        x = self.relu1(x)
        x = self.l3(x)
        return x
        ### END SOLUTION
```

In [ ]:
```
# Continue yourself
```