Tutorial 1 - Artificial Pigeon Brain

Last Lecture

- Biological Neural Networks (BNNs)
- Artificial Neurons Networks (ANNs)
- Forward Pass
- Activation Functions
- Mean Squared Error and Cross-Entropy
- Gradient Descent and Backpropagation
- Nonlinearity

Sample code:

Example: 1-layer ANN with MSE and Gradient Descent

In [32]:
```
import math

# data (first column is the bias term)
x = [[1, 0.1,-0.2],
     [1,-0.1, 0.9],
     [1, 1.2, 0.1],
     [1, 1.1, 1.5]]

# labels (desired output)
t = [0, 0, 0, 1]

# initial weights
w = [1, -1, 1]

iterations = 50
learning = 10

def simple_ann_MSE(x, w, t, iterations, learning):

    E = []

    #iterate over epochs
    for ii in range(iterations):
        err = []
        y = []
        #iterate over all the samples x
        for n in range(len(x)):
            v = 0
            # compute w.x
            for p in range(len(x[0])):
                v = v + x[n][p]*w[p]

            #sigmoidal activation
            y.append(1 / (1 + math.e**(-v)))

            #MSE classification error
            err.append((y[n]-t[n])**2)

            #gradient descent to compute new weights
            for p in range(len(w)):
                d = x[n][p]*(y[n]-t[n])*(1-y[n])*(y[n])
                w[p] = w[p] - learning*d

        #sum up classification error
        E.append(sum(err)/len(x))

    return (y, w, E)

(y, w, E) = simple_ann_MSE(x, w, t, iterations, learning)
```

In [33]:
```
print(y)
```

```
[0.0006555859880223744, 0.052128479309599574, 0.04486688739303947, 0.9484229735074757]
```

Example: 1-layer ANN with Cross-Entropy and Gradient Descent

In [34]:
```
def simple_ann_CE(x, w, t, iterations, learning):

    E = []

    #iterate over epochs
    for ii in range(iterations):
        err = []
        y = []

        #iterate over all the samples x
        for n in range(len(x)):
            v = 0

            #compute w.x
            for p in range(len(x[0])):
                v = v + x[n][p]*w[p]

            #sigmoidal activation
            y.append(1 / (1 + math.e**(-v)))

            #cross-entropy classification error
            err.append(-t[n]*math.log(y[n]+ 0.000001) - (1-t[n])*math.log(1-y[n]+ 0.000001))

            #gradient descent to compute new weights
            for p in range(len(w)):
                d = x[n][p]*(y[n]-t[n]) #cross_entropy
                w[p] = w[p] - learning*d

        #sum up classification error
        E.append(sum(err))

    return (y, w, E)

(y, w, E) = simple_ann_CE(x, w, t, iterations, learning)
```

In [35]:
```
print(y)
print(E)
```

```
[1.5220611182206673e-12, 9.152891744525407e-08, 0.0002045758396744526 3, 0.9995046474331183]
[0.35054613953713 15, 6.8053767054762 92, 26.564614243221254, 4.524869189402009, 0.005815929325684 57,
0.004531305013205282, 0.003740286734600861 7, 0.003200165906455406, 0.002806587130395687 4, 0.00250656
51988698794, 0.002270126814312 9998, 0.002078962576250003, 0.001921222789220056 7, 0.001788884353026 1
722, 0.001676313402896 0054, 0.001579433529908 0592, 0.001495220046114977 5, 0.001421380017390883 5, 0.0
013561428005760547, 0.001298118861471155, 0.001246202175971414 2, 0.001199501246758573 4, 0.001157289 3
800816282, 0.001118968213772658, 0.001084040542818441 3, 0.001052089784338982 8, 0.001022764259793738
2, 0.000995765023182587 8, 0.000970836334098440 5, 0.000947758127445412 8, 0.000926340007318987 3, 0.000
906416416336601 6, 0.000887842720123644 8, 0.000870492010603998 9, 0.000854252478517229 8, 0.000839025 24
01733997, 0.000824722529291096, 0.000811266184239666 2, 0.000798586375810865 1, 0.000786620532019308 4,
0.000775312425189770 9, 0.000764611393434522, 0.000754471673974801 4, 0.000744851829985886 4, 0.0007357
142559979797, 0.000727024749566 1308, 0.000718752139070773 5, 0.000710867959247777 6, 0.000703346167448
8022, 0.000696162894798183 7]
```

Exploring the Iris dataset

Let us modify the above code to work with the iris data set.

To begin, load the iris data into Google Colab. Last time we had some difficulty with this, so it's suggested that you use Chrome or Chromium. Another approach to load the iris data is shown below:

In [36]:
```
# use sklearn.datasets to load iris data
from sklearn.datasets import load_iris
features, labels = load_iris(return_X_y=True)
```

The iris data has 150 samples spread across three classes:

1. Iris-setosa,
2. Iris-versicolor,
3. Iris-virginica.

There are three features used:

1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm

To keep things simple, let us pick two of the classes and perform binary classification with our sample code. We will select **Iris-setosa** and **Iris-versicolor** to start:

In [37]:
```
features
```

Out[37]:

In [38]:
```
labels
```

Out[38]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

In [39]:
```
# classification iris-setosa and iris-versicolor
import numpy as np

indices = np.array(range(0,100))

#setup x matrix
x = np.zeros((len(indices), 4 + 1))
x[:,1:5] = features[indices,:]

#add bias column
x[:,0] = np.ones(len(indices))

#labels
t = labels[indices]

# initial weights
w = np.random.rand(5)

iterations = 100
learning = 0.0001

(y, w, E) = simple_ann_CE(x, w, t, iterations, learning)
```

In [40]:
```
for i in range(len(t)):
  print(t[i], y[i])
```

We're able to successfully classify iris-setosa and iris-versicolor.

In [41]:
```
# classification iris-versicolor and iris virginica
indices = np.array(range(50,150))

#setup x matrix
x = np.zeros((len(indices), 4 + 1))
x[:,1:5] = features[indices,:]

#add bias column
x[:,0] = np.ones(len(indices))

#labels
t = labels[indices]-1

# initial weights
w = np.random.rand(5)

iterations = 100
learning = 0.0001


(y, w, E) = simple_ann_CE(x, w, t, iterations, learning)
```

In [42]:
```
for i in range(len(t)):
  print(t[i], y[i])
```

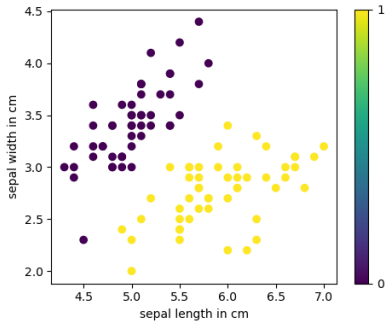
The performance on the iris-versicolor and iris virginica is not as good. To find out why, we will try to visualize the data.

Visualize Iris Dataset

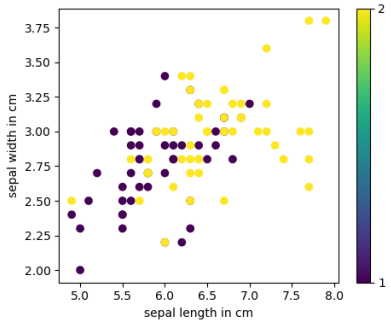
Since the Iris dataset has only 4 inputs we can try to visualize it on a 2-dimensional plane to get a better idea of what is happening.

In [43]:
```
#scatter plot of iris-setosa and iris-versicolor
import numpy as np
from matplotlib import pyplot as plt

indices = np.array(range(0,100))

selected_features = features[indices,:]
selected_labels = labels[indices]

feature_name = ['sepal length in cm', 'sepal width in cm', 'petal length in cm', 'petal width in cm']

x_index = 0
y_index = 1

plt.figure(figsize=(5, 4))
plt.scatter(selected_features[:,x_index], selected_features[:,y_index], c= selected_labels)
plt.colorbar(ticks=[0, 1, 2])
plt.xlabel(feature_name[x_index])
```

```
plt.ylabel(feature_name[y_index])

plt.tight_layout()
plt.show()
```



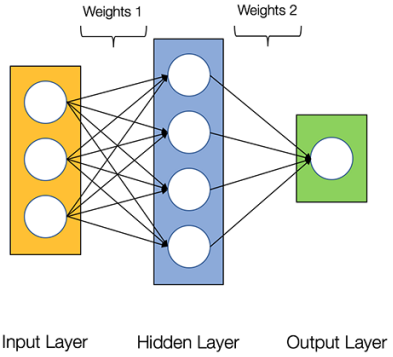
In [44]:
```
# scatter plot of iris-versicolor and iris virginica
import numpy as np
from matplotlib import pyplot as plt

indices = np.array(range(50,150))

selected_features = features[indices,:]
selected_labels = labels[indices]

feature_name = ['sepal length in cm', 'sepal width in cm', 'petal length in cm', 'petal width in cm']

x_index = 0
y_index = 1

plt.figure(figsize=(5, 4))
plt.scatter(selected_features[:,x_index], selected_features[:,y_index], c= selected_labels)
plt.colorbar(ticks=[0, 1, 2])
plt.xlabel(feature_name[x_index])
plt.ylabel(feature_name[y_index])

plt.tight_layout()
plt.show()
```



Nonlinear Separation

Our 1-layer ANN was only successful one of the binary classification combinations. A 1-layer ANN is unable to handle nonlinear separations (or decisions boundaries). To address this we can introduce a second layer known as a hidden layer. How could we do this?

We can just include an additional 1-layer networks as shown in the image below.



We would follow the same process as with the 1-layer network:

1. write out the equations for the forward pass
2. error term can stay the same MSE or Cross-Entropy
3. Gradient descent would be applied now to two layers of weights

First we would consider the forward pass for a 2-layer ANN. We could use a sigmoidal (logistic) activation function to keep things consistent with our earlier example. Note that the activation function will be applied once on the hidden layer, and also on the output layer.

Computing the gradient with respect to the different layers of weights will become more difficult, but still manageable. There are just some additional terms in the chain rule. The second layer weights will be almost identical to what we computed for a 1-layer network, except the input will be the hidden layer activation.

Instead of spending the time to compute gradients with each change to the network, which can be a fun mathematical exercise, we will instead focus on using the PyTorch libraries which handle all of this internally.

(Optional) Develop a 2-layer ANN

Build a 2-layer network using cross-entropy. Determine the gradients with resepect to the layer 1 and layer 2 weights. How could you validate if the gradients were computed correctly?

In [45]:
```
#write code to build a 2-layer network using cross-entropy
```

What is PyTorch?

PyTorch is a scientific computing package that builds on the NumPy library to allow for the use of GPUs. It incorporates deep learning capabilities while maximizing flexibility and speed.

PyTorch Basics

To use PyTorch you must first import the library

In [46]:
```
import torch
```

Tensors

Tensors are n-dimensional arrays that allow that can be used with a GPU to accelerate computing. There are several ways to work with Tensors:

In [47]:
```
# initialize a random tensor
x = torch.rand(4, 3)
print(x)
```

```
tensor([[0.1586, 0.5632, 0.9553],
        [0.3398, 0.8207, 0.3767],
        [0.0399, 0.6654, 0.2312],
        [0.2026, 0.8096, 0.2680]])
```

In [48]:
```
# initialize a tensor loaded with zeros
x = torch.zeros(4, 3)
print(x)
```

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

In [49]:
```
# initialized with data entered manually
x = torch.tensor([2.1, 4.0, -5.2])
print(x)
```

```
tensor([ 2.1000,  4.0000, -5.2000])
```

In [50]:
```
# initialized with data from numPy
import numpy as np
data = np.array([2.1, 4.0, -5.2])
print(data)
x = torch.tensor(data)
print(x)

# note you can easily convert fron tensor to numpy
x_np = x.numpy()
print(x_np)
```

```
[ 2.1  4.  -5.2]
tensor([ 2.1000,  4.0000, -5.2000], dtype=torch.float64)
[ 2.1  4.  -5.2]
```

Tensor size and shape

In [51]:
```
# obtain size of tensor data structure
x = torch.zeros(4, 3)
print(x.size())
```

```
torch.Size([4, 3])
```

Operations

In [52]:
```
# tensor addition
x = torch.rand(4, 3)
y = torch.ones(4, 3)
print(x + y)
```

```
tensor([[1.1469, 1.3182, 1.1060],
        [1.3990, 1.7895, 1.1591],
        [1.7879, 1.9417, 1.9232],
        [1.3624, 1.4250, 1.9854]])
```

In [53]:
```
# tensor multiplication
x = torch.rand(4, 3)
y = torch.ones(4, 3)
print(x * y)
```

```
tensor([[0.5122, 0.5186, 0.2666],
        [0.3253, 0.3971, 0.6395],
        [0.4085, 0.7135, 0.8055],
        [0.9625, 0.6867, 0.8840]])
```

In [54]:
```
# Provide output tensor as argument
result = torch.ones(4,3)
torch.add(x, y, out = result)
print(result)
```

```
tensor([[1.5122, 1.5186, 1.2666],
        [1.3253, 1.3971, 1.6395],
        [1.4085, 1.7135, 1.8055],
        [1.9625, 1.6867, 1.8840]])
```

In [55]:
```
# resize and reshape tensors
x = torch.randn(4, 3)
y = x.view(12)
z = x.view(-1, 4)  # the size -1 is inferred from other dimensions
print(x.size(), y.size(), z.size())
```

```
torch.Size([4, 3]) torch.Size([12]) torch.Size([3, 4])
```

In [56]:
```
# convert one element tensor to a Python number
x = torch.randn(1)
print(x)
print(x.item())
```

```
tensor([1.0855])
1.0854592323303223
```

Automatic Differentiation

The PyTorch autograd package allows for easy computation of derivative. This is handled automatically using a define-by-run framework, which works as you write your code.

To enable this feature you need to set the Tensor attribute .required_grat to True, at which point it begins to track all operations performed on it. After your computation have been completed you can call .backward() and all the gradients will be computed for you. The gradient for each tensor will be stored in the tensor attribute .grad.

Each tensor also has a .grad_fn attribute which references Function that has created it.

In [57]:
```
# Example computation of gradients
x = torch.rand(4, 3, requires_grad=True)
print(x)

#perform some operations
y = x + 10
z = y*y
out = z.mean()

print(z, out)
```

```
tensor([[0.8549, 0.5004, 0.0251],
        [0.0303, 0.9263, 0.6100],
        [0.3137, 0.5454, 0.9576],
        [0.8399, 0.7917, 0.3600]], requires_grad=True)
tensor([[117.8289, 110.2586, 100.5022],
        [100.6079, 119.3836, 112.5725],
        [106.3724, 111.2051, 120.0686],
        [117.5030, 116.4615, 107.3302]], grad_fn=<MulBackward0>) tensor(111.6745, grad_fn=<MeanBackward0>)
```

In [58]:
```
# compute the gradient with respect to the output
out.backward()
print(x.grad)
```

```
tensor([[1.8092, 1.7501, 1.6708],
        [1.6717, 1.8210, 1.7683],
        [1.7190, 1.7576, 1.8263],
        [1.8066, 1.7986, 1.7267]])
```

Artificial Neural Networks in PyTorch

In this example we will train an "artificial pigeon" to perform a digit recognition task. That is, we will use the MNIST dataset of hand-written digits, and train the pigeon to **recognize a small digit, namely a digit that is less than 3**. This problem is a **binary classification problem** we want to predict which of two classes an input image is a part of.

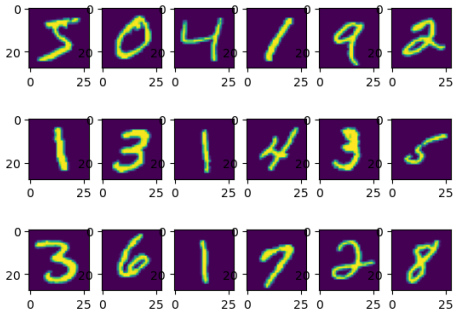
1) Load MNIST Data

The MNIST dataset contains hand-written digits that are 28x28=784 pixels large. Here are a few digits in the dataset:

In [59]:
```
from torchvision import datasets, transforms

# Load the data
mnist_train = datasets.MNIST('data', train=True, download=True)
mnist_train = list(mnist_train)[:2000]
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to data\MNIST\raw\train-images-idx3-ubyte.gz
  0%|          | 0/9912422 [00:00<?, ?it/s]
Extracting data\MNIST\raw\train-images-idx3-ubyte.gz to data\MNIST\raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to data\MNIST\raw\train-labels-idx1-ubyte.gz
  0%|          | 0/28881 [00:00<?, ?it/s]
Extracting data\MNIST\raw\train-labels-idx1-ubyte.gz to data\MNIST\raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to data\MNIST\raw\t10k-images-idx3-ubyte.gz
  0%|          | 0/1648877 [00:00<?, ?it/s]
Extracting data\MNIST\raw\t10k-images-idx3-ubyte.gz to data\MNIST\raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to data\MNIST\raw\t10k-labels-idx1-ubyte.gz
  0%|          | 0/4542 [00:00<?, ?it/s]
Extracting data\MNIST\raw\t10k-labels-idx1-ubyte.gz to data\MNIST\raw
```

In [60]:
```
# plot the first 18 images in the training data
for k, (image, label) in enumerate(mnist_train[:18]):
    plt.subplot(3, 6, k+1)
    plt.imshow(image)
```



2) Defining the ANN Forward Pass

Here is an implementation of the artificial pigeon brain in PyTorch. Don't worry if this code or the explanations don't make sense yet.

In [61]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
import matplotlib.pyplot as plt # for plotting

import torch.optim as optim

torch.manual_seed(1) # set the random seed

class Pigeon(nn.Module):
    def __init__(self):
        super(Pigeon, self).__init__()
        self.layer1 = nn.Linear(784, 30) # 784 = 28x28 every pixel is a feature

        self.layer2 = nn.Linear(30, 1)
    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        activation1 = self.layer1(flattened)
        activation1 = F.relu(activation1)
        activation2 = self.layer2(activation1)
        return activation2

pigeon = Pigeon()
```

In this network, there are 28x28 = 784 input neurons, to work with our 28x28 pixel images. We have a single output neuron and a hidden layer of 30 neurons.

The variable `pigeon.layer1` contains information about the connectivity between the input layer and the hidden layer (stored as a matrix), and the biases (stored as a vector).

Similarly, the variable `pigeon.layer2` contains information about the weights between the hidden layer and the output layer, and the bias.

The weights and biases adjust during training, so they are called the model's **parameters**.

In [62]:
```
# view parameters
for w in pigeon.layer1.parameters():
    print(w)
```

```
Parameter containing:
tensor([[ 0.0184, -0.0158, -0.0069,  ...,  0.0068, -0.0041,  0.0025],
        [-0.0274, -0.0224, -0.0309,  ..., -0.0029,  0.0013, -0.0167],
        [ 0.0282, -0.0095, -0.0340,  ..., -0.0141,  0.0056, -0.0335],
        ...,
        [ 0.0267,  0.0186, -0.0326,  ...,  0.0047, -0.0072, -0.0301],
        [-0.0190,  0.0291,  0.0221,  ...,  0.0067,  0.0206,  0.0151],
        [ 0.0226,  0.0331,  0.0182,  ...,  0.0150,  0.0278, -0.0073]],
       requires_grad=True)
Parameter containing:
tensor([-0.0210,  0.0144,  0.0214, -0.0018, -0.0185,  0.0275, -0.0284, -0.0248,
        -0.0180, -0.0168, -0.0226, -0.0093,  0.0211, -0.0311,  0.0002, -0.0010,
         0.0201,  0.0176, -0.0050,  0.0118, -0.0089,  0.0078, -0.0072,  0.0118,
         0.0122, -0.0125,  0.0326,  0.0306,  0.0058, -0.0089],
       requires_grad=True)
```

3) Test Network Forward Pass

Here is an example of using the network to classify whether the image contains a small digit.

In [63]:
```
# make predictions for the first 10 images in mnist_train
img_to_tensor = transforms.ToTensor() #transform the image data into a 28x28 matrix of numbers
```

```
for k, (image, label) in enumerate(mnist_train[:10]):
    inval = img_to_tensor(image)
    outval = pigeon(inval)       # find the output activation given input
    prob = torch.sigmoid(outval) # turn the activation into a probability
    print(prob)
```

```
tensor([[0.5009]], grad_fn=<SigmoidBackward0>)
tensor([[0.5219]], grad_fn=<SigmoidBackward0>)
tensor([[0.4872]], grad_fn=<SigmoidBackward0>)
tensor([[0.5012]], grad_fn=<SigmoidBackward0>)
tensor([[0.4940]], grad_fn=<SigmoidBackward0>)
tensor([[0.4966]], grad_fn=<SigmoidBackward0>)
tensor([[0.5027]], grad_fn=<SigmoidBackward0>)
tensor([[0.4993]], grad_fn=<SigmoidBackward0>)
tensor([[0.4996]], grad_fn=<SigmoidBackward0>)
tensor([[0.4942]], grad_fn=<SigmoidBackward0>)
```

Since we haven't trained the network yet, the predicted probability of images containing a small digit is close to half. The "pigeon" is unsure.

In order for the network to be useful, we need to actually train it, so that the weights are actually meaningful, non-random values.

4) Update Parameters using Gradient Descent with Cross-Entropy

To update the parameters (weights) we will first use the network to make predictions, then compare the predictions agains the ground truth. To compare the predictions against actual values we'll compute a classification error using the Cross-Entropy equation.

The classification error, that is how good or bad the prediction was compared to the actual values is more commonly referred to as the **loss** and Cross-Entropy is the **loss function**. The introduction of a loss function makes our problem a **optimization** problem: what set of parameters minimizes the loss across the training examples?

Turning a learning problem into an optimization problem is actually a very subtle but important step in many machine learning tools, because it allows us to use tools from mathematical optimization.

That there are **optimizers** that can tune the network parameters for us is also really, really cool. The gradient descent algorithm we derived in the last lecture is one example of an optimizer.

For now, we will choose a standard loss function for a binary classification problem: the **binary cross-entropy loss**. We'll also choose a **stochastic gradient descent** optimizer. We'll talk about what these mean later in the course.

In [64]:
```
# simplified training code to train `pigeon` on the "small digit recognition" task
import torch.optim as optim
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.SGD(pigeon.parameters(), lr=0.005)
```

Now, we can start to train the pigeon network, similar to the way we would train a real pigeon:

1. We'll show the network pictures of digits, one by one
2. We'll see what the network predicts
3. We'll check the loss function for that example digit, comparing the network prediction against the ground truth
4. We'll make a small update to the parameters to try and improve the loss for that digit
5. We'll continue doing this many times -- let's say 1000 times

For simplicity, we'll use 1000 images, and show the network each image only once.

In [65]:
```
for (image, label) in mnist_train[:1000]:
    # actual ground truth: is the digit less than 3?
    actual = torch.tensor(label < 3).reshape([1,1]).type(torch.FloatTensor)
    # pigeon prediction
    out = pigeon(img_to_tensor(image)) # step 1-2
    # update the parameters based on the loss
    loss = criterion(out, actual)      # step 3
    loss.backward()                    # step 4 (compute the updates for each parameter)
    optimizer.step()                   # step 4 (make the updates for each parameter)
    optimizer.zero_grad()              # a clean up step for PyTorch
```

It is very common to run into errors with changing different data types to tensors with the correct shape.

5) Test Updated Network Forward Pass

In [66]:
```
# make predictions for the first 10 images in mnist_train
for k, (image, label) in enumerate(mnist_train[:10]):
    print(label, torch.sigmoid(pigeon(img_to_tensor(image))))
```

```
5 tensor([[0.0995]], grad_fn=<SigmoidBackward0>)
0 tensor([[0.8375]], grad_fn=<SigmoidBackward0>)
4 tensor([[0.1820]], grad_fn=<SigmoidBackward0>)
1 tensor([[0.6976]], grad_fn=<SigmoidBackward0>)
9 tensor([[0.0347]], grad_fn=<SigmoidBackward0>)
2 tensor([[0.2681]], grad_fn=<SigmoidBackward0>)
1 tensor([[0.2112]], grad_fn=<SigmoidBackward0>)
3 tensor([[0.0804]], grad_fn=<SigmoidBackward0>)
1 tensor([[0.2899]], grad_fn=<SigmoidBackward0>)
4 tensor([[0.0575]], grad_fn=<SigmoidBackward0>)
```

Not bad! We'll use the probability 50% as the cutoff for making a discrete prediction. Then, we can compute the accuracy on the 1000 images we used to train the network.

6)a) Validation of Network on Training Data

In [67]:
```
# computing the error and accuracy on the training set
error = 0
for (image, label) in mnist_train[:1000]:
    prob = torch.sigmoid(pigeon(img_to_tensor(image)))
    if (prob < 0.5 and label < 3) or (prob >= 0.5 and label >= 3):
        error += 1
print("Training Error Rate:", error/1000)
print("Training Accuracy:", 1 - error/1000)
```

```
Training Error Rate: 0.112
Training Accuracy: 0.888
```

The accuracy on those 1000 images is 96%, which is really good considering that we only showed the network each image only once.

However, this accuracy is not representative of how well the network is doing, because the network was *trained* on the data. The network had a chance to see the actual answer, and learn from that answer. To get a better sense of the network's predictive accuracy, we should compute accuracy numbers on a **test set**: a set of images that were not seen in training.

6)b) Validation of Network on Testing Data

In [68]:
```
# computing the error and accuracy on a test set
error = 0
for (image, label) in mnist_train[1000:2000]:
    prob = torch.sigmoid(pigeon(img_to_tensor(image)))
    if (prob < 0.5 and label < 3) or (prob >= 0.5 and label >= 3):
        error += 1
print("Test Error Rate:", error/1000)
print("Test Accuracy:", 1 - error/1000)
```

```
Test Error Rate: 0.137
Test Accuracy: 0.863
```