*ECE-GY 9123 / Spring 2021*

# Homework 1

**Please upload your assignments on or before February 12, 2021**.

- You are encouraged to discuss ideas with each other, you **must acknowledge** your collaborator, and you **must compose your own** writeup and/or code independently.
- We **require** answers to theory questions to be written in LaTeX. Handwritten and scanned submissions will not be graded.
- We require answers to coding questions in the form of a Jupyter notebook. Please also include a brief, coherent explanation of both your code and your results.
- Upload your answers in the form of a **single PDF** on Gradescope.
-

---

0. **(0.5 points)** Introduce yourself on the HW1Q0 thread on Ed Stem! Mention a little bit about your background, interests, and why you wish to learn about neural nets and deep learning.
1. **(1.5 points)** *Fun with vector calculus*. This question has two parts.

   a. If $x$ is a $d$-dimensional vector variable, write down the gradient of the function $f(x) = \|x\|_2^2$.

   Since x is a d-dimensional data, so the gradient of f(x) is

   $$\bigtriangledown f(x) = [\frac{df(x)}{dx_1}, \frac{df(x)}{dx_2}, \dots, \frac{df(x)}{dx_d}] = [2x_1, 2x_2, \dots, 2x_d]$$

   b. Suppose we have $n$ data points are real $d$-dimensional vectors. Analytically derive a constant vector $\mu$ for which the MSE loss function

   $$L(\mu) = \sum_{i=1}^{n} \|x_i - \mu\|_2^2$$

   is minimized.
   we can expand the MSE loss function first.

   $$\begin{aligned} L(\mu) &= \sum_{i=1}^{n} \|x_i - \mu\|_2^2 \\ &= \sum_{i=1}^{n} (x_i - \mu)^T (x_i - \mu) \end{aligned}$$

   Then we can do the gradient part

   $$\frac{dL(\mu)}{d\mu} = \sum_{i=1}^{n} (-2x_i + 2\mu)$$

   Then we need to minimize the loss function, so we set it to 0. $\frac{dL}{d\mu} = 0$
   Therefore we can get

   $$\sum_{i=1}^{n} 2x_i = \sum_{i=1}^{n} 2\mu$$
   $$\text{we can get } \mu = \frac{\sum_{i=1}^{n} x_i}{n}$$

2. **(2 points)** *Linear regression with non-standard losses*. In class we derived an analytical expression for the optimal linear regression model using the least squares loss. If $X$ is the matrix of training data points (stacked row-wise) and $y$ is the vector of labels, then:

a. Using matrix/vector notation, write down a loss function that measures the training error in terms of the $\ell_1$-norm.

The loss function is:

$$\sum_{i=1}^{n} |\hat{y_i} - y_i|$$

The $\hat{y}$ is the predicted values and y is the true values.

b. Can you write down the optimal linear model in closed form? If not, why not?

No. The closed form probably may be good for smaller dataset. However, for larger dataset, $X^T X$ may not exist. If the

linear regression model can be represented by

$$y = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \sum_{j=0}^{m} W^T X$$

where $y$ is the response variable and $X$ may be a m-dimensional sample vector, and $W$ is the weight. When we compute

the $W$, we can get:

$$W = (X^T X)^{-1} X^T y$$

And in a larger data set, $X^T X$ may not exist (the matrix may be non-invertible or singular) and that is why it can not written in closed form.

c. If the answer to b is no, can you think of an alternative algorithm to optimize the loss function? Comment on its pros and cons.

We can use gradient descent to optimize the loss function.

pros: It is computational efficient. It produces a stable error gradient and a stable convergence.

cons: It probably will converge to the local minima. It is hard to decide the learning rate.

3. **(2 points)** *Hard coding a multi-layer perceptron*. The functional form for a single perceptron is given by $y = \mathrm{sign}(\langle w, x \rangle + b)$, where $x$ is the data point and $y$ is the predicted label. Suppose your data is 5-dimensional (i.e., $x = (x_1, x_2, x_3, x_4, x_5)$) and real-valued. Find a simple 2-layer network of perceptrons that implements the *Decreasing-Order* function, i.e., it returns +1 if

$$x_1 > x_2 > x_3 > x_4 > x_5$$

and -1 otherwise. Your network should have 2 layers: the input nodes, feeding into 4 hidden perceptrons, which in turn feed into 1 output perceptron. Clearly indicate all the weights and biases of all the 5 perceptrons in your network.

Since we have five input nodes, $x_1, x_2, x_3, x_4, x_5$,so we can get functions from input nodes to hidden layer, it is like:

$$
\begin{aligned}
a_1 &= w_{11}x_1 + w_{21}x_2 + w_{31}x_3 + w_{41}x_4 + w_{51}x_5 + b_1 = 0 \quad (1)\\
a_2 &= w_{12}x_1 + w_{22}x_2 + w_{32}x_3 + w_{42}x_4 + w_{52}x_5 + b_2 = 0 \quad (2)\\
a_3 &= w_{13}x_1 + w_{23}x_2 + w_{33}x_3 + w_{43}x_4 + w_{53}x_5 + b_3 = 0 \quad (3)\\
a_4 &= w_{14}x_1 + w_{24}x_2 + w_{34}x_3 + w_{44}x_4 + w_{54}x_5 + b_4 = 0 \quad (4)
\end{aligned}
$$

$w_{(1)} = [1, -1, 0, 0, 0]$ and because $x_1$ and $x_2$ are real valued which means we can know the value. so, $b_1 = 0$
$w_{(2)} = [0, 1, -1, 0, 0]$ and because $x_1$ and $x_2$ are real valued which means we can know the value. so, $b_2 = 0$
$w_{(3)} = [0, 0, 1, -1, 0]$ and because $x_1$ and $x_2$ are real valued which means we can know the value. so, $b_3 = 0$
$w_{(4)} = [0, 0, 0, 1, -1]$ and because $x_1$ and $x_2$ are real valued which means we can know the value. so, $b_4 = 0$

We can get $z_1, z_2, z_3, z_4$. If $x_1 > x_2$, then $a_1 > 0$, same as the others $a_i$. If $x_1 <= x_2$, the $a_1 <= 0$ We will put these results to the non-linear function so that we can get $z_1, z_2, z_3, z_4$ Then we can go to next layer which is the output layer. We can get the function as below:

$$w_1 z_1 + w_2 z_2 + w_3 z_3 + w_4 z_4 + b = y \quad (5)$$

So, as long as any function in (1), or (2), or (3), or (4), the result is equal to 0 or less than 0, after our non linear function, it will return the value 0. Otherwise, it will return value 1. So we can set $w_{(5)} = [1, 1, 1, 1]$ and $b = -3.5$

4. **(4 points)** This exercise is meant to introduce you to neural network training using Pytorch. Open the (incomplete) Jupyter notebook provided as an attachment to this homework in Google Colab (or other cloud service of your choice) and complete the missing items. Save your finished notebook in PDF format and upload along with your answers to the above theory questions in a single PDF.

OK, thus far we have been talking about linear models. All these can be viewed as a single-layer neural net. The next step is to move on to multi-layer nets. Training these is a bit more involved, and implementing from scratch requires time and effort. Instead, we just use well-established libraries. I prefer PyTorch, which is based on an earlier library called Torch (designed for training neural nets via backprop).

## Import packages

```
import numpy as np
import torch
import torchvision
```

## Try pytorch

Torch handles data types a bit differently. Everything in torch is a *tensor*.

Randomly generate a, a is an array and when we use torch.from_numpy which will let a become a tensor

```
a = np.random.rand(2,3)
b = torch.from_numpy(a)

# Q4.1 Display the contents of a, b
# ...
# ...
print("a:")
print(a)
print()
print("b:")
print(b)
```

```
a:
[[0.30435482 0.61588698 0.41877524]
 [0.05174126 0.26479272 0.58359857]]
```

```
    b:
    tensor([[0.3044, 0.6159, 0.4188],
            [0.0517, 0.2648, 0.5836]], dtype=torch.float64)
```

The idea in Torch is that tensors allow for easy forward (function evaluations) and backward (gradient) passes.

Here, we need to understand the requires_grad. If there’s a single input to an operation that requires gradient, its output will also require gradient. Conversely, only if all inputs don’t require gradient, the output also won’t require it. So, if the input has requires_grad=True, then the output also has require gradient. Backward computation will not happen when all tensors do not require gradient.

```
A = torch.rand(2,2)
b = torch.rand(2,1)
x = torch.rand(2,1, requires_grad=True)

y = torch.matmul(A,x) + b

print(A)
print(b)
print(x)

print(y)

z = y.sum()
print(z)
z.backward()
print(x.grad)
print(x)
```

```
    tensor([[0.7209, 0.4897],
            [0.8969, 0.5733]])
    tensor([[0.6203],
            [0.2386]])
    tensor([[0.6009],
            [0.9437]], requires_grad=True)
    tensor([[1.5156],
            [1.3186]], grad_fn=<AddBackward0>)
    tensor(2.8342, grad_fn=<SumBackward0>)
```

```
tensor([[1.6178],
        [1.0630]])
tensor([[0.6009],
        [0.9437]], requires_grad=True)
```

## download data

Notice how the backward pass computed the gradients using autograd. OK, enough background. Time to train some networks. Let us load the *Fashion MNIST* dataset, which is a database of grayscale images of clothing items.

```
trainingdata = torchvision.datasets.FashionMNIST('./FashionMNIST/',train=True,download=True,transform=torchvision.transforms.ToTensor
testdata = torchvision.datasets.FashionMNIST('./FashionMNIST/',train=False,download=True,transform=torchvision.transforms.ToTensor())
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to ./FashionMNIST/FashionMNIS

                                        26427392/? [00:20<00:00, 4703814.53it/s]

Extracting ./FashionMNIST/FashionMNIST/raw/train-images-idx3-ubyte.gz to ./FashionMNIST/FashionMNIST/raw
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to ./FashionMNIST/FashionMNIS

                                        32768/? [00:00<00:00, 101419.14it/s]

Extracting ./FashionMNIST/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ./FashionMNIST/FashionMNIST/raw
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to ./FashionMNIST/FashionMNIST


                                        4423680/? [00:17<00:00, 886207.23it/s]

Extracting ./FashionMNIST/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ./FashionMNIST/FashionMNIST/raw
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to ./FashionMNIST/FashionMNIST

0%                                      0/5148 [00:00<?, ?it/s]

Extracting ./FashionMNIST/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ./FashionMNIST/FashionMNIST/raw
Processing...
Done!
/usr/local/lib/python3.6/dist-packages/torchvision/datasets/mnist.py:480: UserWarning: The given NumPy array is not writeable,
  return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
```

## Explore data

Let us examine the size of the dataset. When we print it out, we can see it should have 28*28 features in each data point

```
# Q4.2 How many training and testing data points are there in the dataset?
# What is the number of features in each data point?
# ...
# ...
print(trainingdata)
print(testdata)
print(trainingdata.targets.shape)

print(trainingdata.data.shape)
```

```
Dataset FashionMNIST
    Number of datapoints: 60000
    Root location: ./FashionMNIST/
    Split: Train
    StandardTransform
Transform: ToTensor()
Dataset FashionMNIST
    Number of datapoints: 10000
    Root location: ./FashionMNIST/
    Split: Test
    StandardTransform
Transform: ToTensor()
torch.Size([60000])
torch.Size([60000, 28, 28])
```

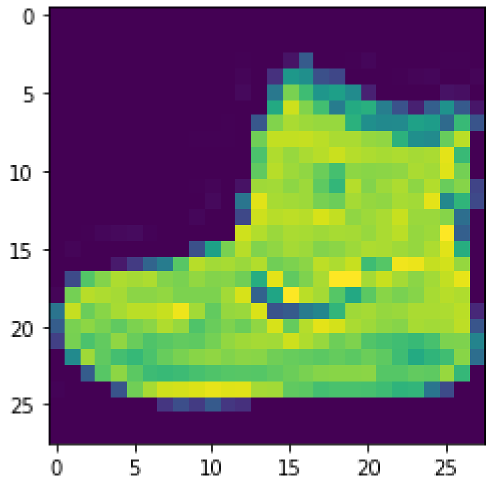
Let us try to visualize some of the images. Since each data point is a tensor (not an array) we need to postprocess to use matplotlib.

```
import matplotlib.pyplot as plt
%matplotlib inline

image, label = trainingdata[0]
```

```
# Q4.3 Assuming each sample is an image of size 28x28, show it in matplotlib.
# ...
# ...
print(label)
plt.imshow(image.resize(28,28))
```

```
9
/usr/local/lib/python3.6/dist-packages/torch/tensor.py:447: UserWarning: non-inplace resize is deprecated
  warnings.warn("non-inplace resize is deprecated")
<matplotlib.image.AxesImage at 0x7fe570243d68>
```



Let's try plotting several images. This is conveniently achieved in PyTorch using a *data loader*, which loads data in batches. We will get 64 data points every data. It will be easy to compute later. For traindata, it will shuffle every time.

```
trainDataLoader = torch.utils.data.DataLoader(trainingdata, batch_size=64, shuffle=True,)
testDataLoader = torch.utils.data.DataLoader(testdata, batch_size=64, shuffle=False)
images, labels = iter(trainDataLoader).next()
print(images.size(), labels)
```

```
torch.Size([64, 1, 28, 28]) tensor([6, 6, 3, 7, 7, 4, 6, 4, 2, 3, 8, 8, 9, 2, 5, 7, 2, 6, 2, 4, 2, 6, 9, 5,
        1, 9, 0, 5, 6, 4, 5, 5, 3, 1, 2, 5, 7, 1, 4, 9, 6, 8, 7, 0, 1, 6, 4, 6,
        3, 1, 6, 7, 5, 5, 8, 6, 6, 2, 6, 9, 6, 2, 8, 1])
```

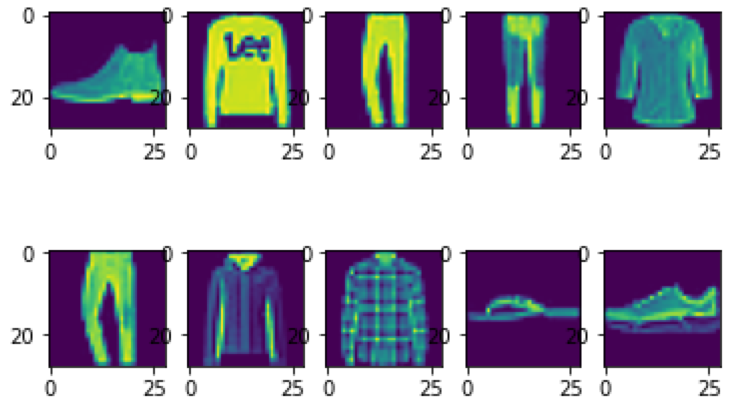
```
print(image.shape)

     torch.Size([1, 28, 28])
```

Get image data from testDataLoader, we ploted first 10 images and we used subplot to organize it.

```
# Q4.4 Visualize the first 10 images of the first minibatch
# returned by testDataLoader.
# ...
# ...
test_images, test_labels = iter(testDataLoader).next()
for i in range(0,10):
  plt.subplot(2,5,i+1)
  plt.imshow(test_images[i].resize(28,28))
```

```
     /usr/local/lib/python3.6/dist-packages/torch/tensor.py:447: UserWarning: non-inplace resize is deprecated
       warnings.warn("non-inplace resize is deprecated")
```



## Define linear model

Now we are ready to define our linear model. Here is some boilerplate PyTorch code that implements the forward model for a single layer network for logistic regression (similar to the one discussed in class notes).

```
class LinearReg(torch.nn.Module):
  def __init__(self):
    super(LinearReg, self).__init__()
    self.linear = torch.nn.Linear(28*28,10)

  def forward(self, x):
    x = x.view(-1,28*28)
    transformed_x = self.linear(x)
    return transformed_x

net = LinearReg().cuda()
Loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)
```

## train network

Cool! Now we have set everything up. Let's try to train the network. We will train the network defined above. We will calculate the average loss every epoch fro training data and test data.

```
train_loss_history = []
test_loss_history = []

# Q4.5 Write down a for-loop that trains this network for 20 epochs,
# and print the train/test losses.
# Save them in the variables above. If done correctly, you should be able to
# execute the next code block.
# ...
# ...

for epoch in range(20):
    tr_loss_list = []
    ts_loss_list = []
    for i, data in enumerate(trainDataLoader,0):
```

```
    for i, data in enumerate(trainDataLoader,0):
      tr_inputs, tr_labels = data
      optimizer.zero_grad()

      outputs = net(tr_inputs.cuda())
      loss = Loss(outputs,tr_labels.cuda())
      loss.backward()
      optimizer.step()
      tr_loss_list.append(loss.item())
      #print(tr_loss_list)
    tr_loss_avg = sum(tr_loss_list)/len(tr_loss_list)
    train_loss_history.append(tr_loss_avg)

    for i, data in enumerate(testDataLoader,0):
      ts_inputs, ts_labels = data
      optimizer.zero_grad()

      outputs = net(ts_inputs.cuda())
      loss = Loss(outputs,ts_labels.cuda())
      loss.backward()
      optimizer.step()
      ts_loss_list.append(loss.item())

    ts_loss_avg = sum(ts_loss_list)/len(ts_loss_list)
    test_loss_history.append(ts_loss_avg)
```

```
print(train_loss_history)
print(test_loss_history)
```

```
[0.9559048773891636, 0.6520971681898845, 0.5895328064526576, 0.5561908759605656, 0.534356366819156, 0.518594569282364, 0.506614
[0.7253690883991825, 0.6322184330338885, 0.590196823997862, 0.56479350244923, 0.5474471246740621, 0.5343411177586598, 0.5241817
```

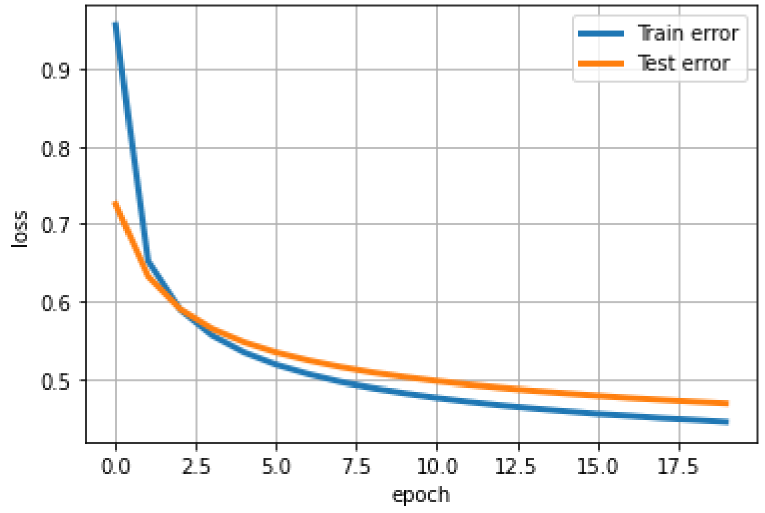
```
print(labels)
```

```
tensor([6, 6, 3, 7, 7, 4, 6, 4, 2, 3, 8, 8, 9, 2, 5, 7, 2, 6, 2, 4, 2, 6, 9, 5,
        1, 9, 0, 5, 6, 4, 5, 5, 3, 1, 2, 5, 7, 1, 4, 9, 6, 8, 7, 0, 1, 6, 4, 6,
        3, 1, 6, 7, 5, 5, 8, 6, 6, 2, 6, 9, 6, 2, 8, 1])
```

plot the loss graph

```
plt.plot(range(20),train_loss_history,'-',linewidth=3,label='Train error')
plt.plot(range(20),test_loss_history,'-',linewidth=3,label='Test error')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.grid(True)
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fe519467ef0>
```



Neat! Now let's evaluate our model accuracy on the entire dataset. The predicted class label for a given input image can computed by looking at the output of the neural network model and computing the index corresponding to the maximum activation. Something like

*predicted_output = net(images) _, predicted_labels = torch.max(predicted_output,1)*

```
predicted_output = net(images.cuda())
print(torch.max(predicted_output, 1))
fit = Loss(predicted_output, labels.cuda())
print(labels)
```

```
torch.return_types.max(
values=tensor([ 7.6511,  6.3200,  3.5632,  5.3376,  8.6754,  7.3957,  5.1442,  7.0970,
         7.0676,  6.8849, 10.9263,  4.4119,  7.8901,  7.0190,  7.9354, 10.3466,
         5.3248,  8.9616,  9.7062,  8.1918,  5.0854,  6.7332, 11.9416,  4.4883,
        12.4314,  6.5984,  6.0975,  4.9972,  6.7973,  8.5018,  5.5331,  7.9429,
         7.0852,  6.2794,  6.0893,  7.5056,  5.3275, 10.5116,  3.3444, 12.8705,
         5.3625,  5.4589,  8.5068, 12.9813, 11.0359,  4.2591,  5.7748,  3.3948,
         5.8395, 13.1109,  7.1317,  7.8050,  6.7632,  5.5268,  4.7925,  8.4738,
         5.5584,  8.0022,  5.7146, 12.0763,  5.2627,  5.7551,  3.3652, 11.4393],
       device='cuda:0', grad_fn=<MaxBackward0>),
indices=tensor([6, 6, 3, 7, 7, 6, 6, 4, 2, 3, 8, 8, 5, 2, 5, 7, 4, 6, 2, 4, 6, 6, 9, 5,
        1, 9, 0, 5, 6, 4, 5, 5, 3, 1, 2, 9, 7, 1, 4, 9, 6, 8, 7, 0, 1, 6, 4, 6,
        3, 1, 4, 7, 5, 5, 8, 2, 6, 2, 6, 9, 6, 2, 8, 1], device='cuda:0'))
tensor([6, 6, 3, 7, 7, 4, 6, 4, 2, 3, 8, 8, 9, 2, 5, 7, 2, 6, 2, 4, 2, 6, 9, 5,
        1, 9, 0, 5, 6, 4, 5, 5, 3, 1, 2, 5, 7, 1, 4, 9, 6, 8, 7, 0, 1, 6, 4, 6,
        3, 1, 6, 7, 5, 5, 8, 6, 6, 2, 6, 9, 6, 2, 8, 1])
```

```
print(fit)
```

```
tensor(0.4271, device='cuda:0', grad_fn=<NllLossBackward>)
```

```
print((torch.max(predicted_output, 1)[1]==labels.cuda()).float().sum()/labels.shape[0])
```

```
tensor(0.8906, device='cuda:0')
```

## Test on the dataset

We will use the model we trained before to test the whole dataset and see the accuracy.

```
def evaluate(dataloader):
  # Q4.6 Implement a function here that evaluates training and testing accuracy.
  # Here, accuracy is measured by probability of successful classification.
  # ...
  # ...
  input, label = iter(dataloader).next()
  pred = net(input.cuda())
  acc = (torch.max(pred,1)[1]==label.cuda()).float().sum()/label.shape[0]
  return acc

print('Train acc = %0.2f, test acc = %0.2f' % (evaluate(trainDataLoader), evaluate(testDataLoader)))
```

```
     Train acc = 0.83, test acc = 0.83
```

Finally, we get the accuracies for training data and testing data are 83% and 83% respectively.