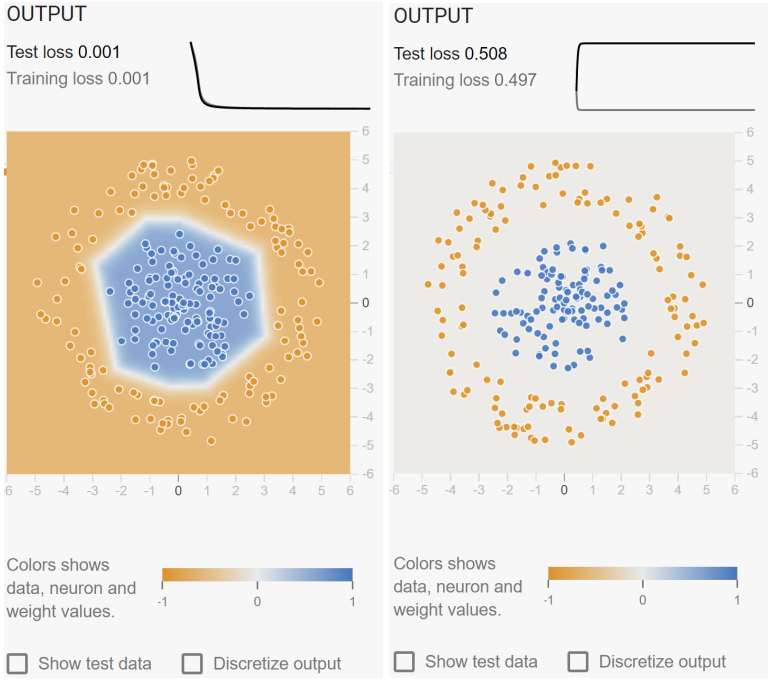
Machine Learning & Data Mining
Caltech CS/CNS/EE 155
Homework 4

Changhao Xu
UID: 2103530
February 4th, 2020

1 Decision Trees

Question A:



The 1st neural network has initial layer weights of random numbers, while the 2nd neural network has initial layer weights of 0. From back propagation, we know that

$$\frac{\partial L}{\partial \mathbf{W}^{(l)}} = \frac{\partial \mathbf{s}^{(l)}}{\partial \mathbf{W}^{(l)}} \times \frac{\partial \mathbf{x}^{(l)}}{\partial \mathbf{s}^{(l)}} \times \frac{\partial \mathbf{s}^{(l+1)}}{\partial \mathbf{x}^{(l)}} \times \frac{\partial \mathbf{x}^{(l+1)}}{\partial \mathbf{s}^{(l+1)}} \times \frac{\partial L}{\partial \mathbf{x}^{(l+1)}}$$

, where

$$\frac{\partial \mathbf{s}^{(l)}}{\partial \mathbf{W}^{(l)}} = \mathbf{x}^{(l-1)}, and \frac{\partial \mathbf{s}^{(l+1)}}{\partial \mathbf{x}^{(l)}} = \mathbf{W}^{(l+1)}$$

. At the last layer

$$\frac{\partial L}{\partial \mathbf{W}^{(l)}} = \frac{\partial \mathbf{s}^{(l)}}{\partial \mathbf{W}^{(l)}} \times \frac{\partial \mathbf{x}^{(l)}}{\partial \mathbf{s}^{(l)}} \times \frac{\partial L}{\partial \mathbf{x}^{(l)}}$$

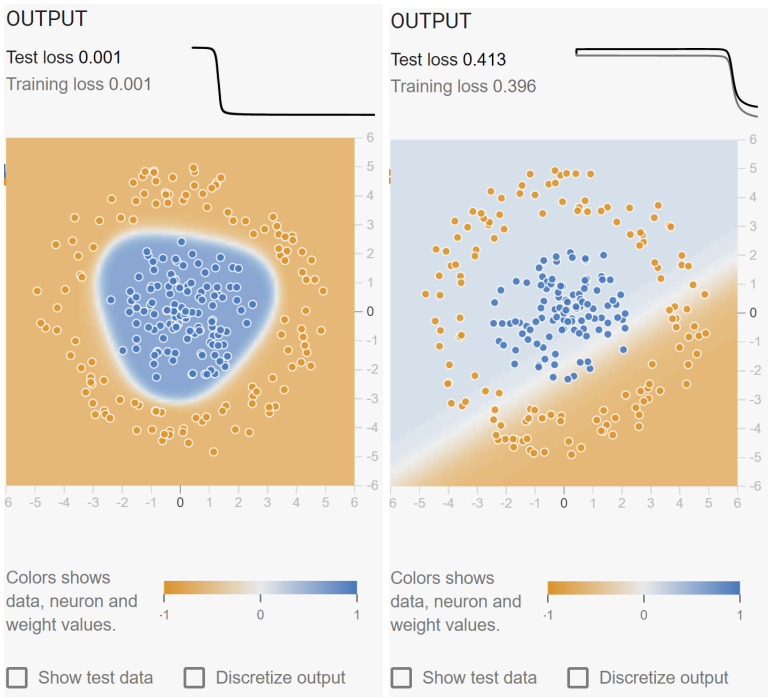
, where

$$\frac{\partial \mathbf{s}^{(l)}}{\partial \mathbf{W}^{(l)}} = \mathbf{x}^{(l-1)}$$

In addition, the activation function ReLU(x) = max(x,0) = 0 when x = 0. Therefore, when the layer weights are initialized with 0, weights can not be updated with gradient descent. As shown in the figures, the 1st neural network performs well, while the 2nd neural network have a high training and test loss.

Question B:

Changhao Xu
UID: 2103530



The 1st neural network has initial layer weights of random numbers, while the 2nd neural network has initial layer weights of 0. From back propagation, we know that

$$\frac{\partial L}{\partial \mathbf{W}^{(l)}} = \frac{\partial \mathbf{s}^{(l)}}{\partial \mathbf{W}^{(l)}} \times \frac{\partial \mathbf{x}^{(l)}}{\partial \mathbf{s}^{(l)}} \times \frac{\partial \mathbf{s}^{(l+1)}}{\partial \mathbf{x}^{(l)}} \times \frac{\partial \mathbf{x}^{(l+1)}}{\partial \mathbf{s}^{(l+1)}} \times \frac{\partial L}{\partial \mathbf{x}^{(l+1)}}$$

, where

$$\frac{\partial \mathbf{s}^{(l)}}{\partial \mathbf{W}^{(l)}} = \mathbf{x}^{(l-1)}, and \frac{\partial \mathbf{s}^{(l+1)}}{\partial \mathbf{x}^{(l)}} = \mathbf{W}^{(l+1)}$$

. What's different is that the activation function Sigmoid(x) = $\frac{exp(x)}{exp(x)+1} \neq 0$ when x = 0. Therefore, starting from the last layer, weights can still be updated since

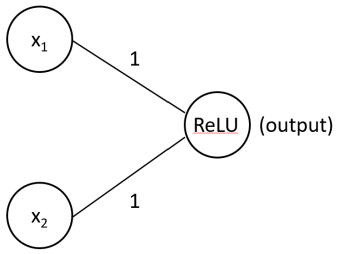
$$\frac{\partial L}{\partial \mathbf{W}^{(l)}} = \frac{\partial \mathbf{s}^{(l)}}{\partial \mathbf{W}^{(l)}} \times \frac{\partial \mathbf{x}^{(l)}}{\partial \mathbf{s}^{(l)}} \times \frac{\partial L}{\partial \mathbf{x}^{(l)}} \neq 0$$

.
Meanwhile, since $\frac{\partial L}{\partial \mathbf{W}^{(l)}}$ is proportional to $\prod \mathbf{W}^{(l)}$, when initial weights are 0, the gradient update is small, making learning speed slow compared with initial weights of random numbers.

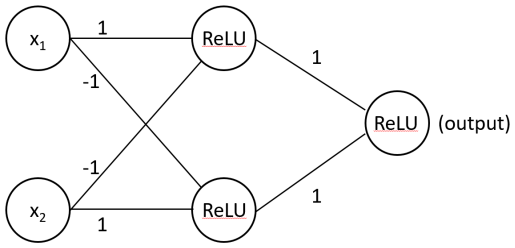
Question C: If we train neural network with ReLU activations using SGD and loop through all the negative examples only, ReLU(x) = max(x,0) = 0 when $x \leq 0$. Then the weights **W** will become 0. In this case, as shown above, weights can not be updated with gradient descent, even with positive examples later.

Question D:

Changhao Xu
UID: 2103530



Question E:



To implement an XOR, minimum 2 layers: an input layer + a hidden layer + output. This is because XOR is not linearly separable, so single layer is not possible.

2 Depth vsWidth on the MNIST Dataset

Question A: torch version: 1.2.0, torchvision version: 0.4.0

Question B: Image height = 28, Image width = 28, no. of channels = 1.
1st index: len = 60000, number of images in the training set
2nd index: len = 2, which represents image (Tensor) and corresponding digit (int)
3rd index: len = 1, which represents no. of channels
4th index: len = 28, image height
5th index: len = 28, image width
There are 60000 images in the training set, and 10000 images in the test set.

Question C: See Jupyter Notebook. Test accuracy = 0.9766.

Question D: See Jupyter Notebook. Test accuracy = 0.9810.

Question E: See Jupyter Notebook. Test accuracy = 0.9830.

Changhao Xu
UID: 2103530

---

3 Convolutional Neural Networks

Question A: Zero-padding scheme benefit: it preserves spatial size, so that the boarder of the image can also be counted. Drawback: We add zeros at the edges, and then the data is changed during convolution. Also we need to do more computations to perform convolutions.

Question B: number of parameters (weights) = $(5 \times 5 \times 3 + 1) \times 8 = 608$.

Question C: shape of the output tensor = $28 \times 28 \times 8$.

Question D: $\begin{bmatrix} 1 & 0.5 \\ 0.5 & 0.25 \end{bmatrix}, \begin{bmatrix} 0.5 & 1 \\ 0.25 & 0.5 \end{bmatrix}, \begin{bmatrix} 0.25 & 0.5 \\ 0.5 & 1 \end{bmatrix}, \begin{bmatrix} 0.5 & 0.25 \\ 1 & 0.5 \end{bmatrix}$

Question E: $\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$

Question F: Pooling will remove the noise of missing pixels, and will smooth the image data taken at various angles/locations.

Question G: See Jupyter Notebook. As shown below, test accuracy = 0.9893.

```
Epoch 1/10:..........
       loss: 0.5945, acc: 0.8859, val loss: 0.0622, val acc: 0.9799
Epoch 2/10:..........
       loss: 0.0962, acc: 0.9719, val loss: 0.0487, val acc: 0.9849
Epoch 3/10:..........
       loss: 0.0805, acc: 0.9772, val loss: 0.0506, val acc: 0.9855
Epoch 4/10:..........
       loss: 0.0744, acc: 0.9796, val loss: 0.0463, val acc: 0.9853
Epoch 5/10:..........
       loss: 0.0694, acc: 0.9803, val loss: 0.0483, val acc: 0.9863
Epoch 6/10:..........
       loss: 0.0669, acc: 0.9810, val loss: 0.0610, val acc: 0.9833
Epoch 7/10:..........
       loss: 0.0663, acc: 0.9817, val loss: 0.0680, val acc: 0.9816
Epoch 8/10:..........
       loss: 0.0630, acc: 0.9831, val loss: 0.0515, val acc: 0.9861
Epoch 9/10:..........
       loss: 0.0649, acc: 0.9822, val loss: 0.0583, val acc: 0.9844
Epoch 10/10:..........
       loss: 0.0612, acc: 0.9837, val loss: 0.0391, val acc: 0.9893
```

Changhao Xu
UID: 2103530

Below is the test accuracy for the 10 dropout probabilities.

```
p = 0.0:..........
        loss: 0.4287, acc: 0.9413, val loss: 0.0652, val acc: 0.9805
p = 0.1111111111111111:..........
        loss: 0.4703, acc: 0.8885, val loss: 0.0521, val acc: 0.9848
p = 0.2222222222222222:..........
        loss: 0.4737, acc: 0.9250, val loss: 0.0784, val acc: 0.9763
p = 0.3333333333333333:..........
        loss: 0.4459, acc: 0.9008, val loss: 0.0772, val acc: 0.9792
p = 0.4444444444444444:..........
        loss: 0.6755, acc: 0.8090, val loss: 0.0953, val acc: 0.9724
p = 0.5555555555555556:..........
        loss: 0.6432, acc: 0.8130, val loss: 0.1249, val acc: 0.9631
p = 0.6666666666666666:..........
        loss: 0.5966, acc: 0.8671, val loss: 0.0955, val acc: 0.9703
p = 0.7777777777777777:..........
        loss: 0.7372, acc: 0.7992, val loss: 0.1612, val acc: 0.9547
p = 0.8888888888888888:..........
        loss: 1.3541, acc: 0.5495, val loss: 0.3817, val acc: 0.9119
p = 1.0:..........
        loss: 2.3027, acc: 0.1084, val loss: 146.6301, val acc: 0.1011
```

Most effective strategy for me is to choose proper regularization strength according to convolution layer complexity. Adding # of filters of convolution layer will accordingly add # of parameters dramatically, so regularization is very important. I used layerwise regularization of dropout and batch norm, and turns out batch norm should be put immediately after convolution Conv2d, and before activation function ReLU. Then I put MaxPool2d and Dropout for additional regularization. In my case, batch norm is the most effective as without it I could not get required test performance.
One problem of validating the hyperparameters here is that we choose the parameters depending on its test set performance. This is kind of data snooping as we are learning from the test set. We should instead use cross-validation in out training set and use testing set only when we have finalized the model.

2_notebook

February 5, 2020

1 Problem 2 Sample Code

This sample code is meant as a guide on how to use PyTorch and how to use the relevant model layers. This not a guide on how to design a network and the network in this example is intentionally designed to have poor performace.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
```

[2]:
```
print(torch.__version__, torchvision.__version__)
```

```
1.2.0 0.4.0
```

1.1 Loading MNIST

The `torchvision` module contains links to many standard datasets. We can load the MNIST dataset into a `Dataset` object as follows:

[3]:
```
train_dataset = datasets.MNIST('./data', train=True, download=True,  #␣
 ↪Downloads into a directory ../data
                               transform=transforms.ToTensor())
test_dataset = datasets.MNIST('./data', train=False, download=False,  # No need␣
 ↪to download again
                              transform=transforms.ToTensor())
```

The `Dataset` object is an iterable where each element is a tuple of (input `Tensor`, target):

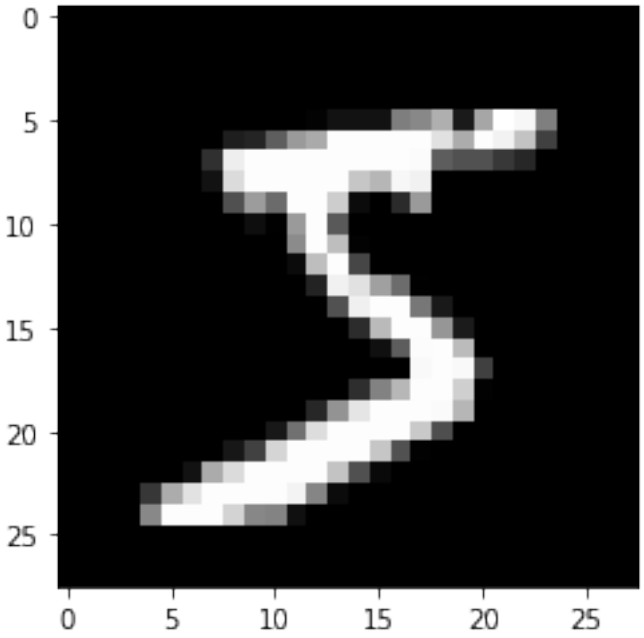
[4]:
```
print(len(test_dataset), type(train_dataset[0][0]), type(train_dataset[0][1]))
print(test_dataset[0][0][0].numpy().shape)
```

```
10000 <class 'torch.Tensor'> <class 'int'>
(28, 28)
```

We can convert images to numpy arrays and plot them with matplotlib:

[5]:
```
plt.imshow(train_dataset[0][0][0].numpy(), cmap='gray')
```

[5]:
```
<matplotlib.image.AxesImage at 0x16939fcb9e8>
```



2 Problem C: Modeling Part 1

2.1 Network Definition

Let's instantiate a model and take a look at the layers.

[6]:
```
model = nn.Sequential(
    # In problem 2, we don't use the 2D structure of an image at all. Our␣
 ↪network
    # takes in a flat vector of the pixel values as input.
    nn.Flatten(),
    nn.Linear(784, 100),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(100, 10)
)
print(model)
```

```
Sequential(
  (0): Flatten()
  (1): Linear(in_features=784, out_features=100, bias=True)
  (2): ReLU()
```

```
  (3): Dropout(p=0.2, inplace=False)
  (4): Linear(in_features=100, out_features=10, bias=True)
)
```

2.2 Training

We also choose an optimizer and a loss function.

[7]:
```
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()
```

We could write our training procedure manually and directly index the `Dataset` objects, but the `DataLoader` object conveniently creates an iterable for automatically creating random mini-batches:

[8]:
```
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=100,␣
 ↪shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=100,␣
 ↪shuffle=True)
```

We now write our backpropagation loop, training for 10 epochs.

[9]:
```
# Some layers, such as Dropout, behave differently during training
model.train()

for epoch in range(10):
    for batch_idx, (data, target) in enumerate(train_loader):
        # Erase accumulated gradients
        optimizer.zero_grad()

        # Forward pass
        output = model(data)

        # Calculate loss
        loss = loss_fn(output, target)

        # Backward pass
        loss.backward()

        # Weight update
        optimizer.step()

    # Track loss each epoch
    print('Train Epoch: %d  Loss: %.4f' % (epoch + 1,  loss.item()))
```

```
Train Epoch: 1  Loss: 0.3003
Train Epoch: 2  Loss: 0.1530
Train Epoch: 3  Loss: 0.1756
Train Epoch: 4  Loss: 0.2292
Train Epoch: 5  Loss: 0.1147
Train Epoch: 6  Loss: 0.1230
```

```
Train Epoch: 7  Loss: 0.1243
Train Epoch: 8  Loss: 0.0426
Train Epoch: 9  Loss: 0.0463
Train Epoch: 10  Loss: 0.0410
```

2.3 Testing

We can perform forward passes through the network without saving gradients.

[10]:
```
# Putting layers like Dropout into evaluation mode
model.eval()

test_loss = 0
correct = 0

# Turning off automatic differentiation
with torch.no_grad():
    for data, target in test_loader:
        output = model(data)
        test_loss += loss_fn(output, target).item()  # Sum up batch loss
        pred = output.argmax(dim=1, keepdim=True)  # Get the index of the max␣
 ↪class score
        correct += pred.eq(target.view_as(pred)).sum().item()

test_loss /= len(test_loader.dataset)

print('Test set: Average loss: %.4f, Accuracy: %d/%d (%.4f)' %
      (test_loss, correct, len(test_loader.dataset),
       100. * correct / len(test_loader.dataset)))
```

```
Test set: Average loss: 0.0008, Accuracy: 9766/10000 (97.6600)
```

3 Problem D: Modeling Part 2

[24]:
```
# nn architecture
model = nn.Sequential(
    # In problem 2, we don't use the 2D structure of an image at all. Our␣
 ↪network
    # takes in a flat vector of the pixel values as input.
    nn.Flatten(),
    nn.Linear(784, 160),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(160, 40),
    nn.ReLU(),
    nn.Dropout(0.1),
    nn.Linear(40, 10)
```

```
)
print(model)
```

```
Sequential(
  (0): Flatten()
  (1): Linear(in_features=784, out_features=160, bias=True)
  (2): ReLU()
  (3): Dropout(p=0.2, inplace=False)
  (4): Linear(in_features=160, out_features=40, bias=True)
  (5): ReLU()
  (6): Dropout(p=0.1, inplace=False)
  (7): Linear(in_features=40, out_features=10, bias=True)
)
```

[25]:
```
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=200,␣
 ↪shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=200,␣
 ↪shuffle=True)
```

[26]:
```
# Some layers, such as Dropout, behave differently during training
model.train()

for epoch in range(12):
    for batch_idx, (data, target) in enumerate(train_loader):
        # Erase accumulated gradients
        optimizer.zero_grad()

        # Forward pass
        output = model(data)

        # Calculate loss
        loss = loss_fn(output, target)

        # Backward pass
        loss.backward()

        # Weight update
        optimizer.step()

    # Track loss each epoch
    print('Train Epoch: %d  Loss: %.4f' % (epoch + 1,  loss.item()))
```

```
Train Epoch: 1  Loss: 0.3061
Train Epoch: 2  Loss: 0.2016
```

```
Train Epoch: 3  Loss: 0.1097
Train Epoch: 4  Loss: 0.1010
Train Epoch: 5  Loss: 0.0748
Train Epoch: 6  Loss: 0.1040
Train Epoch: 7  Loss: 0.1054
Train Epoch: 8  Loss: 0.0735
Train Epoch: 9  Loss: 0.0406
Train Epoch: 10  Loss: 0.0283
Train Epoch: 11  Loss: 0.0484
Train Epoch: 12  Loss: 0.0429
```

[27]:
```python
# Putting layers like Dropout into evaluation mode
model.eval()

test_loss = 0
correct = 0

# Turning off automatic differentiation
with torch.no_grad():
    for data, target in test_loader:
        output = model(data)
        test_loss += loss_fn(output, target).item()  # Sum up batch loss
        pred = output.argmax(dim=1, keepdim=True)  # Get the index of the max␣
 ↪class score
        correct += pred.eq(target.view_as(pred)).sum().item()

test_loss /= len(test_loader.dataset)

print('Test set: Average loss: %.4f, Accuracy: %d/%d (%.4f)' %
      (test_loss, correct, len(test_loader.dataset),
       100. * correct / len(test_loader.dataset)))
```

```
Test set: Average loss: 0.0003, Accuracy: 9810/10000 (98.1000)
```

4 Problem E: Modeling Part 3

[58]:
```python
# nn architecture
model = nn.Sequential(
    # In problem 2, we don't use the 2D structure of an image at all. Our␣
 ↪network
    # takes in a flat vector of the pixel values as input.
    nn.Flatten(),
    nn.Linear(784, 500),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(500, 300),
```

```
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(300, 100),
    nn.ReLU(),
    nn.Dropout(0.1),
    nn.Linear(100, 100),
    nn.ReLU(),
    nn.Dropout(0.05),
    nn.Linear(100, 10)
)
print(model)
```

```
Sequential(
  (0): Flatten()
  (1): Linear(in_features=784, out_features=500, bias=True)
  (2): ReLU()
  (3): Dropout(p=0.3, inplace=False)
  (4): Linear(in_features=500, out_features=300, bias=True)
  (5): ReLU()
  (6): Dropout(p=0.2, inplace=False)
  (7): Linear(in_features=300, out_features=100, bias=True)
  (8): ReLU()
  (9): Dropout(p=0.1, inplace=False)
  (10): Linear(in_features=100, out_features=100, bias=True)
  (11): ReLU()
  (12): Dropout(p=0.05, inplace=False)
  (13): Linear(in_features=100, out_features=10, bias=True)
)
```

[59]:
```
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=800,␣
 ↪shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=800,␣
 ↪shuffle=True)
```

[60]:
```
# Some layers, such as Dropout, behave differently during training
model.train()

for epoch in range(20):
    for batch_idx, (data, target) in enumerate(train_loader):
        # Erase accumulated gradients
        optimizer.zero_grad()

        # Forward pass
        output = model(data)
```

```
        # Calculate loss
        loss = loss_fn(output, target)

        # Backward pass
        loss.backward()

        # Weight update
        optimizer.step()

    # Track loss each epoch
    print('Train Epoch: %d  Loss: %.4f' % (epoch + 1,  loss.item()))
```

```
Train Epoch: 1  Loss: 0.3636
Train Epoch: 2  Loss: 0.1985
Train Epoch: 3  Loss: 0.1427
Train Epoch: 4  Loss: 0.1113
Train Epoch: 5  Loss: 0.1069
Train Epoch: 6  Loss: 0.1313
Train Epoch: 7  Loss: 0.0580
Train Epoch: 8  Loss: 0.0709
Train Epoch: 9  Loss: 0.0573
Train Epoch: 10  Loss: 0.0301
Train Epoch: 11  Loss: 0.0406
Train Epoch: 12  Loss: 0.0660
Train Epoch: 13  Loss: 0.0305
Train Epoch: 14  Loss: 0.0468
Train Epoch: 15  Loss: 0.0352
Train Epoch: 16  Loss: 0.0168
Train Epoch: 17  Loss: 0.0333
Train Epoch: 18  Loss: 0.0321
Train Epoch: 19  Loss: 0.0315
Train Epoch: 20  Loss: 0.0114
```

[61]:
```
# Putting layers like Dropout into evaluation mode
model.eval()

test_loss = 0
correct = 0

# Turning off automatic differentiation
with torch.no_grad():
    for data, target in test_loader:
        output = model(data)
        test_loss += loss_fn(output, target).item()  # Sum up batch loss
        pred = output.argmax(dim=1, keepdim=True)  # Get the index of the max␣
 ↪class score
        correct += pred.eq(target.view_as(pred)).sum().item()
```

```
test_loss /= len(test_loader.dataset)

print('Test set: Average loss: %.4f, Accuracy: %d/%d (%.4f)' %
      (test_loss, correct, len(test_loader.dataset),
       100. * correct / len(test_loader.dataset)))
```

```
Test set: Average loss: 0.0001, Accuracy: 9830/10000 (98.3000)
```

3_notebook

February 7, 2020

1 Problem 3

Use this notebook to write your code for problem 3.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

[2]:
```
import torch

print(torch.cuda.is_available())
print(torch.cuda.current_device())
print(torch.cuda.get_device_name(0))
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
True
0
GeForce GTX 1050 Ti with Max-Q Design
```

1.1 3D - Convolutional network

As in problem 2, we have conveniently provided for your use code that loads and preprocesses the MNIST data.

[3]:
```
# load MNIST data into PyTorch format
import torch
import torchvision
import torchvision.transforms as transforms


# set batch size
batch_size = 32

# load training data downloaded into data/ folder
mnist_training_data = torchvision.datasets.MNIST('data/', train=True,␣
 ↪download=True,
                                                  transform=transforms.ToTensor())
# transforms.ToTensor() converts batch of images to 4-D tensor and normalizes␣
 ↪0-255 to 0-1.0
```

```
training_data_loader = torch.utils.data.DataLoader(mnist_training_data,
                                                   batch_size=batch_size,
                                                   shuffle=True)

# load test data
mnist_test_data = torchvision.datasets.MNIST('data/', train=False,␣
 ↪download=True,
                                             transform=transforms.ToTensor())
test_data_loader = torch.utils.data.DataLoader(mnist_test_data,
                                               batch_size=batch_size,
                                               shuffle=False)
```

[4]:
```
# look at the number of batches per epoch for training and validation
print(f'{len(training_data_loader)} training batches')
print(f'{batch_size} samples in each batch')
print(f'{len(training_data_loader) * batch_size} total training samples')
print(f'{len(test_data_loader)} validation batches')
```

```
1875 training batches
32 samples in each batch
60000 total training samples
313 validation batches
```

[5]:
```
# sample model
import torch.nn as nn

model = nn.Sequential(
    nn.Conv2d(1, 70, kernel_size=(3,3)),
    nn.BatchNorm2d(70),
    nn.ReLU(),
    nn.MaxPool2d((2,2)),
    nn.Dropout(p=0.2),

    nn.Conv2d(70, 70, kernel_size=(3,3)),
    nn.BatchNorm2d(70),
    nn.ReLU(),
    nn.MaxPool2d((2,2)),
    nn.Dropout(p=0.2),

    nn.Flatten(),
    nn.Linear(25*70, 70),
    nn.ReLU(),
    nn.Linear(70, 10)
    # PyTorch implementation of cross-entropy loss includes softmax layer
)
```

[6]:
```
# why don't we take a look at the shape of the weights for each layer
for p in model.parameters():
```

```
    print(p.data.shape)
```

```
torch.Size([70, 1, 3, 3])
torch.Size([70])
torch.Size([70])
torch.Size([70])
torch.Size([70, 70, 3, 3])
torch.Size([70])
torch.Size([70])
torch.Size([70])
torch.Size([70, 1750])
torch.Size([70])
torch.Size([10, 70])
torch.Size([10])
```

[7]:
```
# our model has some # of parameters:
count = 0
for p in model.parameters():
    n_params = np.prod(list(p.data.shape)).item()
    count += n_params
print(f'total params: {count}')
```

```
total params: 168430
```

[8]:
```
# For a multi-class classification problem
import torch.optim as optim
criterion = nn.CrossEntropyLoss()
optimizer = optim.RMSprop(model.parameters())
```

2 Final test accuracy when trained for 10 epochs

[9]:
```
# Train the model for 10 epochs, iterating on the data in batches
n_epochs = 10

# store metrics
training_accuracy_history = np.zeros([n_epochs, 1])
training_loss_history = np.zeros([n_epochs, 1])
validation_accuracy_history = np.zeros([n_epochs, 1])
validation_loss_history = np.zeros([n_epochs, 1])

for epoch in range(n_epochs):
    print(f'Epoch {epoch+1}/10:', end='')
    train_total = 0
    train_correct = 0
    # train
```

```
    model.train()
    for i, data in enumerate(training_data_loader):
        images, labels = data
        optimizer.zero_grad()
        # forward pass
        output = model(images)
        # calculate categorical cross entropy loss
        loss = criterion(output, labels)
        # backward pass
        loss.backward()
        optimizer.step()

        # track training accuracy
        _, predicted = torch.max(output.data, 1)
        train_total += labels.size(0)
        train_correct += (predicted == labels).sum().item()
        # track training loss
        training_loss_history[epoch] += loss.item()
        # progress update after 180 batches (~1/10 epoch for batch size 32)
        if i % 180 == 0: print('.',end='')
    training_loss_history[epoch] /= len(training_data_loader)
    training_accuracy_history[epoch] = train_correct / train_total
    print(f'\n\tloss: {training_loss_history[epoch,0]:0.4f}, acc:␣
↪{training_accuracy_history[epoch,0]:0.4f}',end='')

    # validate
    test_total = 0
    test_correct = 0
    with torch.no_grad():
        model.eval()
        for i, data in enumerate(test_data_loader):
            images, labels = data
            # forward pass
            output = model(images)
            # find accuracy
            _, predicted = torch.max(output.data, 1)
            test_total += labels.size(0)
            test_correct += (predicted == labels).sum().item()
            # find loss
            loss = criterion(output, labels)
            validation_loss_history[epoch] += loss.item()
        validation_loss_history[epoch] /= len(test_data_loader)
        validation_accuracy_history[epoch] = test_correct / test_total
    print(f', val loss: {validation_loss_history[epoch,0]:0.4f}, val acc:␣
↪{validation_accuracy_history[epoch,0]:0.4f}')
```

```
Epoch 1/10:...
```

```
        loss: 0.5945, acc: 0.8859, val loss: 0.0622, val acc: 0.9799
Epoch 2/10:...
        loss: 0.0962, acc: 0.9719, val loss: 0.0487, val acc: 0.9849
Epoch 3/10:...
        loss: 0.0805, acc: 0.9772, val loss: 0.0506, val acc: 0.9855
Epoch 4/10:...
        loss: 0.0744, acc: 0.9796, val loss: 0.0463, val acc: 0.9853
Epoch 5/10:...
        loss: 0.0694, acc: 0.9803, val loss: 0.0483, val acc: 0.9863
Epoch 6/10:...
        loss: 0.0669, acc: 0.9810, val loss: 0.0610, val acc: 0.9833
Epoch 7/10:...
        loss: 0.0663, acc: 0.9817, val loss: 0.0680, val acc: 0.9816
Epoch 8/10:...
        loss: 0.0630, acc: 0.9831, val loss: 0.0515, val acc: 0.9861
Epoch 9/10:...
        loss: 0.0649, acc: 0.9822, val loss: 0.0583, val acc: 0.9844
Epoch 10/10:...
        loss: 0.0612, acc: 0.9837, val loss: 0.0391, val acc: 0.9893
```

3 Test accuracy for the 10 dropout probabilities

[12]:
```python
p_s = np.linspace(0, 1, num = 10)

for p in p_s:
    model = nn.Sequential(
        nn.Conv2d(1, 64, kernel_size=(3,3)),
        nn.BatchNorm2d(64),
        nn.ReLU(),
        nn.MaxPool2d((2,2)),
        nn.Dropout(p= p),

        nn.Conv2d(64, 64, kernel_size=(3,3)),
        nn.BatchNorm2d(64),
        nn.ReLU(),
        nn.MaxPool2d((2,2)),
        nn.Dropout(p= p),

        nn.Flatten(),
        nn.Linear(25*64, 64),
        nn.ReLU(),
        nn.Linear(64, 10)
        # PyTorch implementation of cross-entropy loss includes softmax layer
    )
    print(f'p = {p}:', end='')
    criterion = nn.CrossEntropyLoss()
```

```
    optimizer = optim.RMSprop(model.parameters())
    # store metrics
    training_accuracy = 0
    training_loss = 0
    validation_accuracy = 0
    validation_loss = 0

    train_total = 0
    train_correct = 0
    # train
    model.train()
    for i, data in enumerate(training_data_loader):
        images, labels = data
        optimizer.zero_grad()
        # forward pass
        output = model(images)
        # calculate categorical cross entropy loss
        loss = criterion(output, labels)
        # backward pass
        loss.backward()
        optimizer.step()

        # track training accuracy
        _, predicted = torch.max(output.data, 1)
        train_total += labels.size(0)
        train_correct += (predicted == labels).sum().item()
        # track training loss
        training_loss += loss.item()
        # progress update after 180 batches
        if i % 180 == 0: print('.',end='')
    training_loss /= len(training_data_loader)
    training_accuracy = train_correct / train_total
    print(f'\n\tloss: {training_loss:0.4f}, acc: {training_accuracy:0.
↪4f}',end='')

    # validate
    test_total = 0
    test_correct = 0
    with torch.no_grad():
        model.eval()
        for i, data in enumerate(test_data_loader):
            images, labels = data
            # forward pass
            output = model(images)
            # find accuracy
            _, predicted = torch.max(output.data, 1)
            test_total += labels.size(0)
```

```
            test_correct += (predicted == labels).sum().item()
            # find loss
            loss = criterion(output, labels)
            validation_loss += loss.item()
        validation_loss /= len(test_data_loader)
        validation_accuracy = test_correct / test_total
    print(f', val loss: {validation_loss:0.4f}, val acc: {validation_accuracy:0.
↪4f}')
```

```
p = 0.0:...
        loss: 0.4287, acc: 0.9413, val loss: 0.0652, val acc: 0.9805
p = 0.1111111111111111:...
        loss: 0.4703, acc: 0.8885, val loss: 0.0521, val acc: 0.9848
p = 0.2222222222222222:...
        loss: 0.4737, acc: 0.9250, val loss: 0.0784, val acc: 0.9763
p = 0.3333333333333333:...
        loss: 0.4459, acc: 0.9008, val loss: 0.0772, val acc: 0.9792
p = 0.4444444444444444:...
        loss: 0.6755, acc: 0.8090, val loss: 0.0953, val acc: 0.9724
p = 0.5555555555555556:...
        loss: 0.6432, acc: 0.8130, val loss: 0.1249, val acc: 0.9631
p = 0.6666666666666666:...
        loss: 0.5966, acc: 0.8671, val loss: 0.0955, val acc: 0.9703
p = 0.7777777777777777:...
        loss: 0.7372, acc: 0.7992, val loss: 0.1612, val acc: 0.9547
p = 0.8888888888888888:...
        loss: 1.3541, acc: 0.5495, val loss: 0.3817, val acc: 0.9119
p = 1.0:...
        loss: 2.3027, acc: 0.1084, val loss: 146.6301, val acc: 0.1011
```

Above, we output the training loss/accuracy as well as the validation loss and accuracy. Not bad! Let's see if you can do better.