Lab 8: Training and testing with the from-scratch library

Author: Nigel Nelson

Course: CS-3450

Introduction:

- This lab builds upon past labs, where the forward and backward propagation operations were defined for a set of network layers. This lab combines this past work to build a simple neural network that is then trained on the fashion-MNIST data set in order to prove that the underlying mathematical functions are operating correctly.

In [1]:
```
import torch
import numpy as np
import matplotlib.pyplot as plt
import warnings
import os.path
```

In [2]:
```
import network
import layers
```

warnings.filterwarnings('ignore') # If you see warnings that you know you can ignore, it can be useful to enable this.

In [3]:
```
EPOCHS = 40
# For simple regression problem
TRAINING_POINTS = 1000
```

In [4]:
```
# For fashion-MNIST and similar problems
DATA_ROOT = '/data/cs3450/data/'
FASHION_MNIST_TRAINING = '/data/cs3450/data/fashion_mnist_flattened_training.
FASHION_MNIST_TESTING = '/data/cs3450/data/fashion_mnist_flattened_testing.np
CIFAR10_TRAINING = '/data/cs3450/data/cifar10_flattened_training.npz'
CIFAR10_TESTING = '/data/cs3450/data/cifar10_flattened_testing.npz'
CIFAR100_TRAINING = '/data/cs3450/data/cifar100_flattened_training.npz'
CIFAR100_TESTING = '/data/cs3450/data/cifar100_flattened_testing.npz'
```

In [5]:
```
# With this block, we don't need to set device=DEVICE for every tensor.
torch.set_default_dtype(torch.float32)
if torch.cuda.is_available():
     torch.cuda.set_device(0)
     torch.set_default_tensor_type(torch.cuda.FloatTensor)
     print("Running on the GPU")
else:
     print("Running on the CPU")
```

```
Running on the GPU
```

In [6]:
```
def create_linear_training_data():
    """
    This method simply rotates points in a 2D space.
    Be sure to use L2 regression in the place of the final softmax layer befo
    data!
    :return: (x,y) the dataset. x is a torch tensor where columns are trainin
             y is a torch tensor where columns are one-hot labels for the tra
    """
    x = torch.randn((2, TRAINING_POINTS))
    x1 = x[0:1, :].clone()
    x2 = x[1:2, :]
    y = torch.cat((-x2, x1), axis=0)
    return x, y
```

In [7]:
```
def create_folded_training_data():
    """
    This method introduces a single non-linear fold into the sort of data cre
    Be sure to use L2 regression in the place of the final softmax layer befo
    data!
    :return: (x,y) the dataset. x is a torch tensor where columns are trainin
             y is a torch tensor where columns are one-hot labels for the tra
    """
    x = torch.randn((2, TRAINING_POINTS))
    x1 = x[0:1, :].clone()
    x2 = x[1:2, :]
    x2 *= 2 * ((x2 > 0).float() - 0.5)
    y = torch.cat((-x2, x1), axis=0)
    return x, y
```

In [8]:
```
def create_square():
    """
    This is a square example
    insideness is true if the points are inside the square.
    :return: (points, insideness) the dataset. points is a 2xN array of point
    """
    win_x = [2,2,3,3]
    win_y = [1,2,2,1]
    win = torch.tensor([win_x,win_y],dtype=torch.float32)
    win_rot = torch.cat((win[:,1:],win[:,0:1]),axis=1)
    t = win_rot - win # edges tangent along side of poly
    rotation = torch.tensor([[0, 1],[-1,0]],dtype=torch.float32)
    normal = rotation @ t # normal vectors to each side of poly
        # torch.matmul(rotation,t) # Same thing

    points = torch.rand((2,2000),dtype = torch.float32)
    points = 4*points

    vectors = points[:,np.newaxis,:] - win[:,:,np.newaxis] # reshape to fill
    insideness = (normal[:,:,np.newaxis] * vectors).sum(axis=0)
    insideness = insideness.T
    insideness = insideness > 0
    insideness = insideness.all(axis=1)
    return points, insideness
```

In [9]:
```
def create_patterns():
    """
    I don't remember what sort of data this generates -- Dr. Yoder

    :return: (points, insideness) the dataset. points is a 2xN array of point
    """
    pattern1 = torch.tensor([[1, 0, 1, 0, 1, 0]],dtype=torch.float32).T
    pattern2 = torch.tensor([[1, 1, 1, 0, 0, 0]],dtype=torch.float32).T
    num_samples = 1000

    x = torch.zeros((pattern1.shape[0],num_samples))
    y = torch.zeros((2,num_samples))
    # TODO: Implement with shuffling instead?
    for i in range(0,num_samples):
        if torch.rand(1) > 0.5:
            x[:,i:i+1] = pattern1
            y[:,i:i+1] = torch.tensor([[0,1]],dtype=torch.float32).T
        else:
            x[:,i:i+1] = pattern2
            y[:,i:i+1] = torch.tensor([[1,0]],dtype=torch.float32).T
    return x, y
```

In [10]:
```
def load_dataset_flattened(train=True,dataset='Fashion-MNIST',download=False)
    """
    :param train: True for training, False for testing
    :param dataset: 'Fashion-MNIST', 'CIFAR-10', or 'CIFAR-100'
    :param download: True to download. Keep to false afterwords to avoid unne
    :return: (x,y) the dataset. x is a torch tensor where columns are trainin
             y is a torch tensor where columns are one-hot labels for the tra
    """
    if dataset == 'Fashion-MNIST':
        if train:
            path = FASHION_MNIST_TRAINING
        else:
            path = FASHION_MNIST_TESTING
        num_labels = 10
    elif dataset == 'CIFAR-10':
        if train:
            path = CIFAR10_TRAINING
        else:
            path = CIFAR10_TESTING
        num_labels = 10
    elif dataset == 'CIFAR-100':
        if train:
            path = CIFAR100_TRAINING
        else:
            path = CIFAR100_TESTING
        num_labels = 100
    else:
        raise ValueError('Unknown dataset: '+str(dataset))

    if os.path.isfile(path):
        print('Loading cached flattened data for',dataset,'training' if trair
        data = np.load(path)
        x = torch.tensor(data['x'],dtype=torch.float32)
        y = torch.tensor(data['y'],dtype=torch.float32)
        pass
    else:
        class ToTorch(object):
            """Like ToTensor, only redefined by us for 'historical reasons'""

            def __call__(self, pic):
                return torchvision.transforms.functional.to_tensor(pic)

        if dataset == 'Fashion-MNIST':
            data = torchvision.datasets.FashionMNIST(
                root=DATA_ROOT, train=train, transform=ToTorch(), download=dc
        elif dataset == 'CIFAR-10':
            data = torchvision.datasets.CIFAR10(
                root=DATA_ROOT, train=train, transform=ToTorch(), download=dc
        elif dataset == 'CIFAR-100':
            data = torchvision.datasets.CIFAR100(
                root=DATA_ROOT, train=train, transform=ToTorch(), download=dc
        else:
            raise ValueError('This code should be unreachable because of a pr
        x = torch.zeros((len(data[0][0].flatten()), len(data)),dtype=torch.fl
        for index, image in enumerate(data):
            x[:, index] = data[index][0].flatten()
```

```
        labels = torch.tensor([sample[1] for sample in data])
        y = torch.zeros((num_labels, len(labels)), dtype=torch.float32)
        y[labels, torch.arange(len(labels))] = 1
        np.savez(path, x=x.numpy(), y=y.numpy())
    return x, y
```

In [11]:

```
def test_simple_net_forward():
    """
    Function used to verify that the forward propagation of the network works
    """
    device = torch.device('cpu:0')
    dtype = torch.float64

    x = torch.tensor([[3], [2]], dtype=dtype, device=device)
    W = torch.tensor([[4, 5], [-2, 2], [7, 1]], dtype=dtype, device=device)
    b1 = torch.tensor([[1], [-2], [3]], dtype=dtype, device=device)
    M = torch.tensor([[-4, 5, 3], [-2, 2, 7]], dtype=dtype, device=device)
    b2 = torch.tensor([[-3], [2]], dtype=dtype, device=device)

    x_layer = layers.Input((2,1))

    W_layer = layers.Input((3,2))
    W_layer.set(W)
    b1_layer = layers.Input((3,1))
    b1_layer.set(b1)
    M_layer = layers.Input((2,3))
    M_layer.set(M)
    b2_layer = layers.Input((2,1))
    b2_layer.set(b2)

    linear1 = layers.Linear(x_layer, W_layer, b1_layer)
    relu = layers.ReLU(linear1)
    linear2 = layers.Linear(relu, M_layer, b2_layer)

    net = network.Network()

    net.add(x_layer)
    net.add(linear1)
    net.add(relu)
    net.add(linear2)

    net.forward(x)

    net.layers[-1].accumulate_grad(torch.tensor([[1], [1]], dtype=torch.float
    net.backward()

    print('The expected output is:')
    print(torch.tensor([[-17], [138]]))
    print()
    print('The actual output is:')
    print(net.output)
```

In [12]:

```
if __name__ == '__main__':
    dataset = 'Fashion-MNIST'
    # dataset = 'CIFAR-10'
    # dataset = 'CIFAR-100'

#     x_train, y_train = create_linear_training_data()
#     x_train, y_train = create_folded_training_data()
#     points_train, insideness_train = create_square()
    x_train, y_train = load_dataset_flattened(train=True, dataset=dataset, do
    x_test, y_test = load_dataset_flattened(train=False, dataset=dataset, dow
```

```
Loading cached flattened data for Fashion-MNIST training
Loading cached flattened data for Fashion-MNIST testing
```

Defining Hyper-parameters

In [13]:

```
num_epochs = 10
batch_size = 4
num_hidden_nodes = 24
learing_rate = 0.001
```

Defining the network architecture:

In [14]:

```
# Define input layers
x_layer = layers.Input((x_train.shape[0], batch_size), train=False)
W_layer = layers.Input((num_hidden_nodes, x_train.shape[0]))
W_layer.randomize()
b1_layer = layers.Input((num_hidden_nodes, batch_size))
b1_layer.randomize()
M_layer = layers.Input((y_train.shape[0], num_hidden_nodes))
M_layer.randomize()
b2_layer = layers.Input((y_train.shape[0], batch_size))
b2_layer.randomize()

# Scale the weight matrices
W_layer.output.data *= 0.01
M_layer.output.data *= 0.01

#define meta layers
linear1 = layers.Linear(x_layer, W_layer, b1_layer)
relu = layers.ReLU(linear1)
linear2 = layers.Linear(relu, M_layer, b2_layer)
y_layer = layers.Input((y_train.shape[0], batch_size), train=False)
softmax = layers.Softmax(y_layer, linear2)

# Intialize the network and add its layers in order of execution
net = network.Network()
net.add(x_layer)
net.add(W_layer)
net.add(b1_layer)
net.add(linear1)
net.add(relu)
net.add(M_layer)
net.add(b2_layer)
net.add(linear2)
net.add(softmax)
```

Training on the Fashion-MNIST dataset:

In [15]:
```
epoch_metrics_dict = {
    'training_acc': [],
    'testing_acc': [],
    'training_loss': [],
    'testing_loss': [],
}

for epoch in range(num_epochs):

    # Initialize training metrics to 0
    training_num_correct = 0
    total_training_loss = 0
    testing_num_correct = 0
    total_testing_loss = 0

    # Training Loop
    for i in range(x_train.shape[1]//batch_size):

        # Get the correct locations to reference in the training set
        start_idx = i*batch_size
        end_idx = i*batch_size + batch_size

        complete_true_labels = y_train[:, start_idx : end_idx].reshape(y_trai

        net.layers[-1].actual.set(complete_true_labels)

        input_data = x_train[:, start_idx : end_idx].reshape(x_train.shape[0]

        net.forward(input_data)

        # Collect the loss from the training
        total_training_loss += net.layers[len(net.layers) - 1].output

        # Collect the predicted values
        predictions = torch.argmax(net.layers[len(net.layers) - 1].softmax(),
        true_labels = torch.argmax(y_train[:, start_idx : end_idx].reshape(y_

        training_num_correct += (predictions == true_labels).sum()

        net.backward()
        net.step(learing_rate)

    #Testing Loop
    for i in range(x_test.shape[1]//batch_size):

        # Get the correct locations to reference in the testing set
        start_idx = i*batch_size
        end_idx = i*batch_size + batch_size

        complete_true_labels = y_test[:, start_idx : end_idx].reshape(y_test.

        net.layers[-1].actual.set(complete_true_labels)

        input_data = x_test[:, start_idx : end_idx].reshape(x_test.shape[0],

        net.forward(input_data)
```

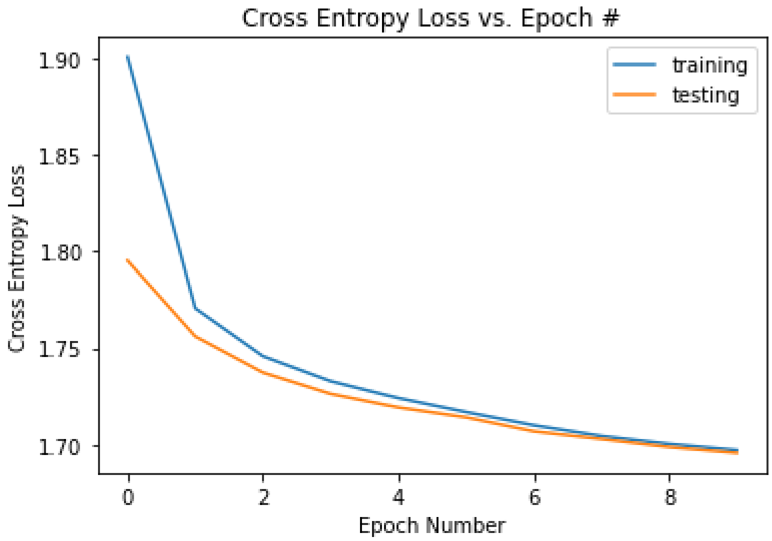
```
        total_testing_loss += net.layers[len(net.layers) - 1].output

        predictions = torch.argmax(net.layers[len(net.layers) - 1].softmax(),
        true_labels = torch.argmax(y_test[:, start_idx : end_idx].reshape(y_t

        testing_num_correct += (predictions == true_labels).sum()


    # Calculate all training/testing metrics and print to console for each ep
    training_loss = total_training_loss.item()/y_train.shape[1]
    testing_loss = total_testing_loss.item()/y_test.shape[1]
    training_acc = training_num_correct/y_train.shape[1]
    testing_acc = testing_num_correct/y_test.shape[1]

    print(f'Epoch #{epoch + 1}:')
    print(f'\tTraining Loss: {training_loss}')
    print(f'\tTesting Loss: {testing_loss}')
    print(f'\tTraining accuracy: {training_acc}')
    print(f'\tTesting accuracy: {testing_acc}')
    print()

    epoch_metrics_dict['training_loss'].append(training_loss)
    epoch_metrics_dict['testing_loss'].append(testing_loss)
    epoch_metrics_dict['training_acc'].append(training_acc)
    epoch_metrics_dict['testing_acc'].append(testing_acc)
```

```
Epoch #1:
	Training Loss: 1.9004578125
	Testing Loss: 1.7952228515625
	Training accuracy: 0.7223333120346069
	Testing accuracy: 0.7993999719619751

Epoch #2:
	Training Loss: 1.770619921875
	Testing Loss: 1.756096875
	Training accuracy: 0.8176000118255615
	Testing accuracy: 0.8166999816894531

Epoch #3:
	Training Loss: 1.7458552083333334
	Testing Loss: 1.737412890625
	Training accuracy: 0.8294000029563904
	Testing accuracy: 0.8252999782562256

Epoch #4:
	Training Loss: 1.732948828125
	Testing Loss: 1.726397265625
	Training accuracy: 0.8364666700363159
	Testing accuracy: 0.8307999968528748

Epoch #5:
	Training Loss: 1.7241739583333333
	Testing Loss: 1.71934296875
	Training accuracy: 0.8415666818618774
	Testing accuracy: 0.833899974822998
```

```
Epoch #6:
        Training Loss: 1.716959765625
        Testing Loss: 1.7142111328125
        Training accuracy: 0.8453500270843506
        Testing accuracy: 0.8377999663352966

Epoch #7:
        Training Loss: 1.7101298177083333
        Testing Loss: 1.706987890625
        Training accuracy: 0.8484166860580444
        Testing accuracy: 0.840999960899353

Epoch #8:
        Training Loss: 1.7045822916666666
        Testing Loss: 1.7030330078125
        Training accuracy: 0.8516333699226379
        Testing accuracy: 0.8425999879837036

Epoch #9:
        Training Loss: 1.7004919270833334
        Testing Loss: 1.698885546875
        Training accuracy: 0.8543500304222107
        Testing accuracy: 0.8452000021934509

Epoch #10:
        Training Loss: 1.69729921875
        Testing Loss: 1.695851953125
        Training accuracy: 0.8565833568572998
        Testing accuracy: 0.8463999629020691
```

Training Curves for Training and Testing Sets:

In [16]:
```
plt.plot(range(num_epochs), epoch_metrics_dict['training_loss'], label='trair
plt.plot(range(num_epochs), epoch_metrics_dict['testing_loss'], label='testir
plt.xlabel('Epoch Number')
plt.ylabel('Cross Entropy Loss')
plt.title('Cross Entropy Loss vs. Epoch #')
plt.legend()
plt.show()
```



In [17]:
```
plt.plot(range(num_epochs), epoch_metrics_dict['training_acc'], label='traini
plt.plot(range(num_epochs), epoch_metrics_dict['testing_acc'], label='testing
plt.xlabel('Epoch Number')
plt.ylabel('Accuracy')
plt.title('Accuracy vs. Epoch #')
plt.legend()
plt.show()
```

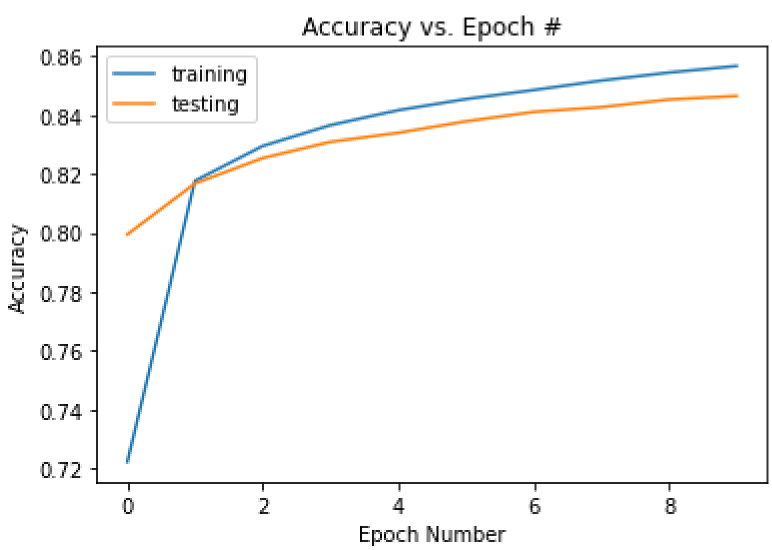


Table Summary of Training Performance:

	Accuracy	Loss (Cross-Entropy)

	Accuracy	Loss (Cross-Entropy)
Training	0.8547	1.6946
Testing	0.8423	1.6865

Discussion:

- This lab sequence has severed as a deep dive in neural networks, and the underlying mathematical functions that allow them to model the data that they are trained on. The first portion of this lab was deriving and testing the forward propagation of our network layers. This was a fairly straight forward experience, yet one of my biggest take-aways from this phase was the importance of verifying the shapes of the matrices that are being operated on. This is because most of the issues in my network were discovered once I realized a pair of matrices did not have the correct shapes. Following this, the back propagation operations for our network layers needed to be derived. From this experience, I re-learned the chain rule for partial derivatives, and through practice, ultimately developed a methodology to derive the derivatives of any network layer. Thanks to the experience I gained working the mathematical operations of the network by hand, creating unit tests for the network and its respective layers was fairly straight forward. With a true understanding of the operations applied at each layer, it was trivial to choose random numbers, work out by hand the expected answers, then use those expected answers in assertions to ensure the layers were behaving properly. Overall, the most frustrating portion of this process was definitely debugging a network that wouldn't train. This is due in large part to the fact that Jupyter notebooks do not have a debugging environment, and as such when the network wasn't training, one had to take educated guesses and simply tamper with the parameters and operations until they saw a change in the outputs. The issues that I ran into resulted in disappearing gradients where my network would no longer train. The first cause was the fact that I had too many hidden nodes, 256 to be exact. When I would attempt to train with this many hidden nodes, my network would train for a single epoch, and then have the gradients diminish to 0 and the loss and accuracy would not change for the remainder of the training session. What I believe happened here is that with a larger batch size, the inner matrices for these hidden nodes were so large that each parameter's partial derivative was such a small fraction of the final loss, that those derivatives were essentially 0, resulting in no change to the parameters epoch to epoch. The second issue that I ran into was that without multiplying my weight vectors by 0.1 or 0.01, my gradients would drop to 0 after a couple epochs. The reason that I believe this step was necessary is that with larger weights in layers with many hidden nodes, the resulting matrix multiplications result in huge numbers, which in turn result in a large loss. This is an issue as subtracting each parameter's partial derivative with respect to a massive loss can in turn cause the existing weights to be wiped out, causing the weight matrices to lose their gradient. This was a difficult issue to debug, but ultimately by scaling the initialization values of the weight matrices, the issue of disappearing gradients was ultimately resolved
- Through all of this knowledge gained, I was ultimately able to get a testing accuracy of **0.846** for the Fashion-MNIST data set. This was done with 24 hidden nodes, a batch size of 4, and a learning rate of 0.001. In order to decrease the likely-hood of overfitting, I trained for only 10 epochs, which is less than half of what I had trained for in earlier lab sequences. Thanks to this approach, my training curves tend to show that there is minimal overfitting occurring in my training. This can be seen in both the loss and accuracy curves, where the training values begin terribly, reflecting the random initialization of weights, but by the 1st epoch achieve

respectable scores. From that first epoch on, the training and testing curves remain reasonably entangled, without one showing drastic increases or decreases that is not mirrored by the other. As such, this is an indication that my network generalized the underlying patterns of the data set with almost 85% accuracy, and minimally memorized the training data set.