EECS663: Homework II

Vimig Socrates

March 2021

1 Problem 1

1.1 Ex. 1

In the 1D case, the loss landscape $C(v)$ is represented as a line shape, with the x-axis being the only variable v_1, and the y-axis being the loss. Therefore, in a simple case of a quadratic curve, our "rolling ball" would roll down to the "bottom" (minimum) of the quadratic.

1.2 Ex. 2

An advantage of online learning is the same as classic SGD, which allows the model the ability to jump out of saddle points and local minima quickly as further iterations of backprop are run. However, when performing backprop on only one sample at a time, this also works against the model. With noisy data, a particular example may adjust the weights incorrectly and throw the backprop algorithm wildly away from a potentially effective local minima. Therefore mini-batching over a subset of the entire data finds a balance between the advantage and disadvantage of SGD.

2 Problem 2

2.1 Ex. 1

If there is only a single neuron that uses a nonsigmoidal activation, then when we compute the backpropagated error for the layer that the nonsigmoidal neuron is on l: δ_j^l, we just need to use the correct derivative of the other activation function f'.

2.2 Ex. 2

In order to validate $\delta_j^L = a_j^L - y_i$, we must compute the two partials with respect to the weights and biases: $\frac{\partial C}{\partial b_j^L} = a_j^L - y_j$ and $\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1}(a_j^L - y_j)$. Starting with the biases:

$$\frac{\partial C}{\partial b_j^L} = \sum_k \frac{\partial C}{\partial z_k^L} \cdot \frac{\partial z_k^L}{\partial b_j^L} \tag{1}$$

Unlike in the original derivation, we are not able to only choose one of the elements in the summation. We find the derivative of the log likelihood cost function wrt the activation to get (we also use the derivative of the softmax function, which is $a_j^L(1 - a_j^L)$):

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L} = \frac{\partial - \ln a_y^L}{\partial a_k^L} = -\frac{1}{a_y^L} \cdot a_j^L(1 - a_j^L) \tag{2}$$

For the jth value of y y_j which is the same as when $j = y$:

$$\delta_j^L = -\frac{1}{a_y^L} \cdot a_j^L(1 - a_j^L) = a_y^L - 1_{j=y} = a_y^L - y_j \tag{3}$$

We can show a similar relationship for the weights term as following:

$$\frac{\partial C}{\partial w_{jk}^L} = \sum_i \frac{\partial C}{\partial z_i^l} \cdot \frac{\partial z_i^l}{\partial w_{jk}^L} = \sum_i \frac{\partial C}{\partial a_i^L} \frac{\partial a_i^L}{\partial z_j^L} \frac{\partial z_j^L}{\partial w_{jk}^L} \tag{4}$$

The first two partials are the same from above, and for we get:

$$\frac{\partial z_j^L}{\partial w_{jk}^L} = \frac{\partial \sum_i w_{ij}^L a_i^{L-1} + b_j^L}{\partial w_{jk}^L} \tag{5}$$

Since only the weight of the respective partial applies (i.e. partials where $i \neq j = 0$), we can simplify to:

$$((w^{l+1})^T \delta^{l+1}) = a_k^{L-1} \tag{6}$$

So put all together, we get what we expect:

$$\frac{\partial C}{\partial w_{jk}^L} \quad = \quad a_k^{L-1}(a_j^L - y_j) \tag{7}$$

Therefore, we've shown that we get the expected backprop of the weights and biases using $\delta_j^L = a_j^L - y_j$.

2.3 Ex. 3

In backprop with linear neurons, we can replace the $\sigma(z) = z$ and $\sigma'(z) = 1$. This makes the following questions:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) = \frac{\partial C}{\partial a_j^L}1 = \frac{\partial C}{\partial a_j^L} \tag{8}$$

This translates to:

$$\delta^L = \nabla_a C \odot \sigma'(z^L) = \nabla_a C \tag{9}$$

Also, to backpropagate the rest of the errors:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) = ((w^{l+1})^T \delta^{l+1}) \tag{10}$$

3 Problem 3

3.1 Ex. 1

In the second equation, we run into significant issues if either $y = 0$ or $y = 1$ because either the left or the right term (respectively) is undefined, as $\ln 0 = undef$. This issue doesn't afflict the first equation, since the model will almost never predict with 100% confidence that a prediction is either 0 or 1. If it does, we would have the same issue, however this is unlikely to be the case. Moreover, we want only one of the sides of the + to hold if our class label is either 0 or 1. In the second equation, this would not be the case, both sides of the equation would impact the loss (incorrectly).

3.2 Ex. 2

In order to determine if C is minimized, we have to take the partial derivative with respect to the output z like so:

$$C = -\frac{1}{n}\sum_x [y \ln \hat{y} + (1-y) ln(1-\hat{y})] \tag{11}$$

We assume that the $\hat{y} = \sigma(z)$:

$$\frac{\partial C}{\partial z} = -y\frac{1}{\sigma(z)}\sigma'(z) + (1-y)\frac{1}{1-\sigma(z)}\sigma'(z) \tag{12}$$

$$\sigma'(z) = \frac{-y}{\sigma(z)} + \frac{(1-y)}{1-\sigma(z)} \tag{13}$$

Set it equal to 0 to find the extrema:

$$\sigma'(z) = \frac{\partial C}{\partial z} = \frac{-y}{\sigma(z)} + \frac{(1-y)}{1-\sigma(z)} \tag{14}$$

$$0 = \frac{-y}{\sigma(z)} + \frac{(1-y)}{1-\sigma(z)} \tag{15}$$

$$\frac{y}{\sigma(z)} = \frac{(1-y)}{1-\sigma(z)} \tag{16}$$

$$y(1-\sigma(z)) = \sigma(z)(1-y) \tag{17}$$

$$y = \sigma(z) \tag{18}$$

Then we must take the second derivative to make sure that we have a minima:

$$\frac{\partial^2 C}{\partial z^2} = \sigma''(z) = \frac{y}{\sigma(z)^2} + \frac{(1-y)}{(1-\sigma(z))^2} \tag{19}$$

$$\tag{20}$$

Since we know that both $1 > \sigma(z) > 0$ and $1 > z > 0$, we know that the above equation is ≥ 1 so our extrema is a minima.

3.3 Ex. 3

We must first perform the forward pass for both the hidden neurons:

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 = \tag{21}$$

$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 = 0.3775 \tag{22}$$

$$net_{h1} = 0.2 * 0.05 + 0.3 * 0.1 + 0.35 = 0.39 \tag{23}$$

$$\tag{24}$$

Squash them both with the sigmoid:

$$out_{h1} = \frac{1}{1-e^{-0.3775}} = 0.59327 \tag{25}$$

$$out_{h2} = \frac{1}{1-e^{-0.39}} = 0.596282 \tag{26}$$

We repeat this with the inputs from this past section for the two output nodes.

$$out_{o1} = \frac{1}{1+e^{-1.105905967}} = 0.75137 \tag{27}$$

$$out_{o2} = \frac{1}{1-e^{-0.39}} = 0.77287 \tag{28}$$

Now we try and calculate the error based on cross-entropy:

$$E_{o1} = 0.01 * \ln(0.75137) + (1 - 0.01) * \ln(1 - 0.75137) * 1.38073 \tag{29}$$
$$= 1.905327 \tag{30}$$
$$E_{o2} = 0.99 * \ln(0.77287) + (1 - 0.99) * \ln(1 - 0.77287) * 1.38073 \tag{31}$$
$$= 0.2755336 \tag{32}$$
$$E_{tot} = 0.2755336 + 1.905327 = 2.1808636 \tag{33}$$

In order to backprop, we can use the equations for the partial derivatives of the cross-entropy loss function wrt the weight and bias terms found in the Nielsen book:

$$\frac{\partial C}{\partial w_j} = x_j(\sigma(z) - y) \tag{34}$$
$$\frac{\partial C}{\partial b} = (\sigma(z) - y) \tag{35}$$

Using the numbers we have, we get:

$$\frac{\partial C}{\partial w_5} = 0.59327 * (0.75137 - 0.01) = 0.4398 \tag{36}$$
$$\frac{\partial C}{\partial w_6} = 0.596282 * (0.75137 - 0.01) = 0.44207 \tag{37}$$
$$\frac{\partial C}{\partial w_7} = 0.59327 * (0.77287 - 0.99) = -0.1288 \tag{38}$$
$$\frac{\partial C}{\partial w_8} = 0.596282 * (0.77287 - 0.99) = -0.1295 \tag{39}$$
$$\frac{\partial C}{\partial b2_{o1}} = (0.75137 - 0.01) = 0.74137 \tag{40}$$
$$\frac{\partial C}{\partial b2_{o2}} = (0.77287 - 0.99) = -0.21713 \tag{41}$$
$$\tag{42}$$

However, for the **hidden neurons**, we will need an additional application of the chain rule to take into consideration the errors from both outputs in the final layer that a particular weight in the hidden layer has of impacting. Therefore:

$$\frac{\partial C}{\partial w_1^1} = \frac{\partial C}{\partial z_1^1}\frac{\partial z_1^1}{\partial w_1^1} \tag{43}$$

$$\frac{\partial C}{\partial w_2^1} = \frac{\partial C}{\partial z_1^1}\frac{\partial z_1^1}{\partial w_2^1} \tag{44}$$

$$\frac{\partial C}{\partial w_3^1} = \frac{\partial C}{\partial z_2^1}\frac{\partial z_2^1}{\partial w_3^1} \tag{45}$$

$$\frac{\partial C}{\partial w_4^1} = \frac{\partial C}{\partial z_2^1}\frac{\partial z_2^1}{\partial w_4^1} \tag{46}$$

$$\tag{47}$$

In the above, z_j^1 is the weighted input sum at hidden unit j. In order to get this partial, we have to sum over both the output errors, like so:

$$\frac{\partial C}{\partial z_j^1} = \sum_{i=1}^{2} \frac{\partial C}{\partial z_i}\frac{\partial z_i}{\partial a_j}\frac{\partial a_j}{\partial z_j^1} \tag{48}$$

$$= \sum_{i=1}^{2} (\sigma(z) - y)(w_{ji})(a_j(1 - a_j)) \tag{49}$$

This gets us:

$$\frac{\partial C}{\partial w_j^1} = \sum_{i=1}^{2} (\sigma(z) - y)(w_{ji})(a_j(1 - a_j))(i_k) \tag{50}$$

where a_j is the activation from the respective hidden neuron, w_{ji} is the edge leading to the respective output node from the particular hidden node, and i_k is the respective input example value.

Plugging in numbers, we have:

$$\frac{\partial C}{\partial w_1^1} = \Big((0.75137 - 0.01) * (0.4) * (0.59327 * (1 - 0.59327)) * (0.05)\Big) + \tag{51}$$

$$\Big((0.77287 - 0.99) * (0.45) * (0.59327 * (1 - 0.59327))(0.05)\Big) = 0.002399$$

$$\frac{\partial C}{\partial w_2^1} = \Big((0.75137 - 0.01) * (0.5) * (0.596282 * (1 - 0.596282)) * (0.05)\Big) + \tag{52}$$

$$\Big((0.77287 - 0.99) * (0.55) * (0.596282 * (1 - 0.596282))(0.05)\Big) = 0.003024$$

$$\frac{\partial C}{\partial w_3^1} = \Big((0.75137 - 0.01) * (0.4) * (0.59327 * (1 - 0.59327)) * (0.1)\Big) + \tag{53}$$

$$\Big((0.77287 - 0.99) * (0.45) * (0.59327 * (1 - 0.59327))(0.1)\Big) = 0.004798$$

$$\frac{\partial C}{\partial w_4^1} = \Big((0.75137 - 0.01) * (0.5) * (0.596282 * (1 - 0.596282)) * (0.1)\Big) + \tag{54}$$

$$\Big((0.77287 - 0.99) * (0.55) * (0.596282 * (1 - 0.596282))(0.1)\Big) = 0.00605 \tag{55}$$

Biases are similar:

$$\frac{\partial C}{\partial b_1^1} = \frac{\partial C}{\partial z_1^1}\frac{\partial z_1^1}{\partial b_1^1} \tag{56}$$

$$\frac{\partial C}{\partial b_2^1} = \frac{\partial C}{\partial z_2^1}\frac{\partial z_2^1}{\partial b_2^1} \tag{57}$$

$$\tag{58}$$

Which leads to:

$$\frac{\partial C}{\partial b_1^1} = \Big((0.75137 - 0.01) * (0.4) * (0.59327 * (1 - 0.59327))\Big) + \tag{59}$$

$$\Big((0.77287 - 0.99) * (0.45) * (0.59327 * (1 - 0.59327))\Big) = 0.04798$$

$$\frac{\partial C}{\partial b_2^1} = \Big((0.75137 - 0.01) * (0.5) * (0.596282 * (1 - 0.596282))\Big) + \tag{60}$$

$$\Big((0.77287 - 0.99) * (0.55) * (0.596282 * (1 - 0.596282))\Big) = 0.06049 \tag{61}$$

4 Bonus

If we consider the following definition of softmax:

$$a_j^L = \frac{e^{cz_j^L}}{\sum_k e^{cz_k^L}} \tag{62}$$

where, conventionally, $c = 1$, but instead make $c \to \infty$, we approach argmax. The max of a_j^L is 1, and everything else is 0. Therefore, with $c = 1$, this is a "soft" version of the argmax function, where other values in the vector are still influential, instead of hard binary values.

Problem_4

March 17, 2021

1 Problem 4

[1]:
```
%load_ext autoreload
%autoreload 2
```

[2]:
```
import torch
import numpy as np
from torchvision import datasets
```

[3]:
```
from six.moves import urllib
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
urllib.request.install_opener(opener)
```

1.1 Load Data

[4]:
```
# ####################################
# import data
# ####################################
# download the MNIST dataset
mnist_trainset = datasets.MNIST(root='data', train=True, download=True,␣
 ↪transform=None)
mnist_testset = datasets.MNIST(root='data', train=False, download=True,␣
 ↪transform=None)
```

[5]:
```
# separate into data and labels
# training data
# reducing training dataset to 1000 points and test dataset to 2000 points in␣
 ↪order to
# create an overfitting model on which to study regularization

train_data = mnist_trainset.data.to(dtype=torch.float32)[:1000]
train_data = train_data.reshape(-1, 784)
train_labels = mnist_trainset.targets.to(dtype=torch.long)[:1000]

print("train data shape: {}".format(train_data.size()))
```

```
print("train label shape: {}".format(train_labels.size()))

# testing data
test_data = mnist_testset.data.to(dtype=torch.float32)[:2000]
test_data = test_data.reshape(-1, 784)
test_labels = mnist_testset.targets.to(dtype=torch.long)[:2000]

print("test data shape: {}".format(test_data.size()))
print("test label shape: {}".format(test_labels.size()))
```

```
train data shape: torch.Size([1000, 784])
train label shape: torch.Size([1000])
test data shape: torch.Size([2000, 784])
test label shape: torch.Size([2000])
```

[6]:
```
# load into torch datasets
train_dataset = torch.utils.data.TensorDataset(train_data, train_labels)
test_dataset = torch.utils.data.TensorDataset(test_data, test_labels)
```

1.2 Set Hyperparameters

[7]:
```
# ###################################
# set hyperparameters
# ###################################
# Parameters
learning_rate = 0.01
num_epochs = 1000 # training for a long time to see overfitting
batch_size = 128

# Network Parameters
n_hidden_1 = 128  # 1st layer number of neurons
n_hidden_2 = 128  # 2nd layer number of neurons
n_hidden_3 = 128  # 3rd layer number of neurons
num_input = 784  # MNIST data input (img shape: 28*28)
num_classes = 10  # MNIST total classes (0-9 digits)
```

1.3 Create Data Loaders

[8]:
```
# ###################################
# creata dataloader
# ###################################
trainloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,
                                          shuffle=True)
testloader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size,
                                          shuffle=False)
```

1.4 Compare Loss Functions

[105]:
```
from prob4 import *
```

[106]:
```
loss = torch.nn.CrossEntropyLoss()
ce_metrics, ce_model = train(trainloader, train_data, train_labels, test_data,␣
 ↪test_labels,
     num_epochs, num_input, num_classes, learning_rate, loss)
```

```
train acc: 12.2  test acc: 10.95         at epoch: 0
train acc: 72.3  test acc: 53.949999999999996    at epoch: 100
train acc: 90.5  test acc: 68.0  at epoch: 200
train acc: 95.19999999999999     test acc: 71.35000000000001     at epoch: 300
train acc: 96.3  test acc: 72.95         at epoch: 400
train acc: 96.89999999999999     test acc: 74.65         at epoch: 500
train acc: 97.1  test acc: 75.14999999999999     at epoch: 600
train acc: 97.6  test acc: 75.9  at epoch: 700
train acc: 97.7  test acc: 76.6  at epoch: 800
train acc: 98.1  test acc: 77.14999999999999     at epoch: 900
```

[107]:
```
loss = torch.nn.MSELoss(reduction="mean")
mse_metrics, mse_model = train(trainloader, train_data, train_labels,␣
 ↪test_data, test_labels,
     num_epochs, num_input, num_classes, learning_rate, loss, mse=True)
```

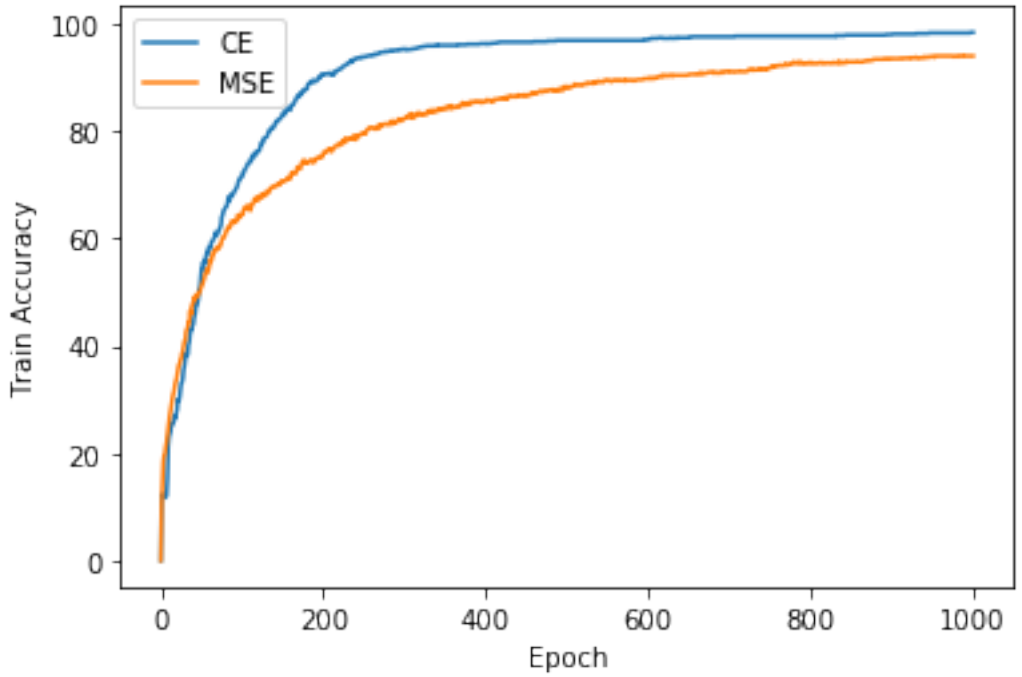
```
train acc: 10.299999999999999    test acc: 8.85  at epoch: 0
train acc: 64.5  test acc: 43.75         at epoch: 100
train acc: 76.0  test acc: 52.400000000000006    at epoch: 200
train acc: 82.5  test acc: 57.85         at epoch: 300
train acc: 85.7  test acc: 60.85         at epoch: 400
train acc: 88.2  test acc: 63.55         at epoch: 500
train acc: 89.9  test acc: 66.3  at epoch: 600
train acc: 91.10000000000001     test acc: 67.85         at epoch: 700
train acc: 92.7  test acc: 69.05         at epoch: 800
train acc: 93.5  test acc: 69.95         at epoch: 900
```

1.4.1 Plot results

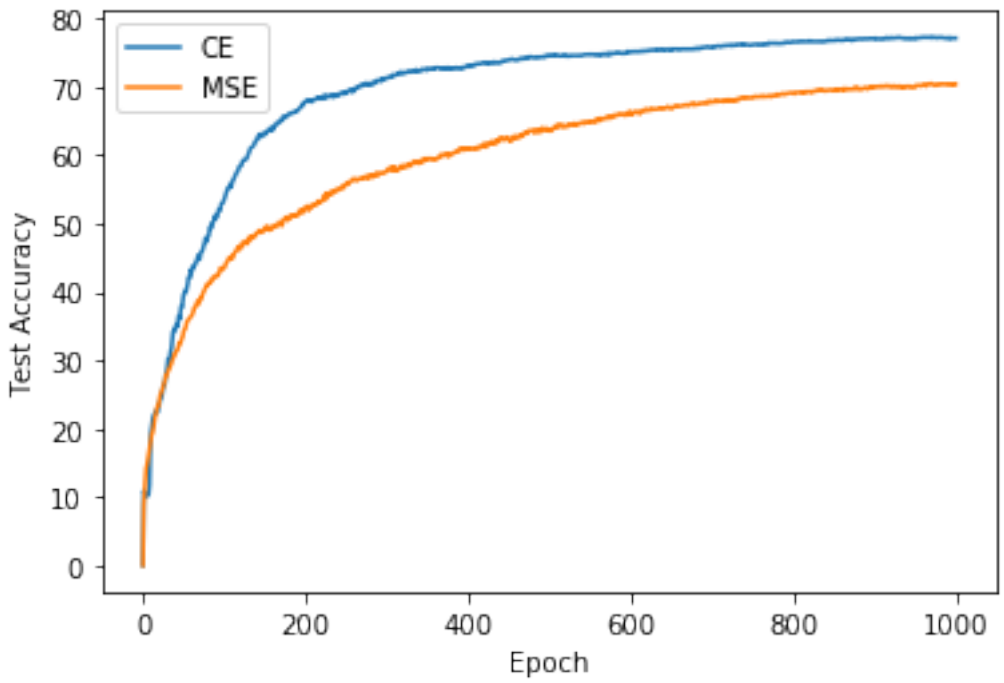
[108]:
```
import matplotlib.pyplot as plt
```

[138]:
```
plt.plot(ce_metrics[:,0], label="CE")
plt.plot(mse_metrics[:,0], label="MSE")
plt.xlabel("Epoch")
plt.ylabel("Train Accuracy")
plt.legend()

plt.show()
```



[110]:
```
plt.plot(ce_metrics[:,1], label="CE")
plt.plot(mse_metrics[:,1], label="MSE")
plt.xlabel("Epoch")
plt.ylabel("Test Accuracy")
plt.legend()
plt.show()
```



Cross entropy converges faster than MSE, as shown by the steeper curve. It also providers a higher accuracy, both in training and test. This is because CE is better suited for a classification task such as MNIST, while MSE is more meant for regression tasks. One of the main reasons for this is that CE heavily penalizes incorrect guesses of the target, while MSE does not. This is what we want in a classification task, as we either have the right target class or not. However, in a regression problem, the answer may be close along the range of values we are predicting on. Moreover, the CE models the error as drawn from a multinomial distribution, as opposed to the Normal distribution modeled by MSE.

1.5 Compare Regularization

1.5.1 L1 Regularization

[111]:
```
from prob4 import *
```

[112]:
```
loss = torch.nn.CrossEntropyLoss()
l1_001_metrics, l1_001_model = train(trainloader, train_data, train_labels,␣
↪test_data, test_labels,
     num_epochs, num_input, num_classes, learning_rate, loss,␣
↪regularization="l1", reg_lambda=0.001)
```

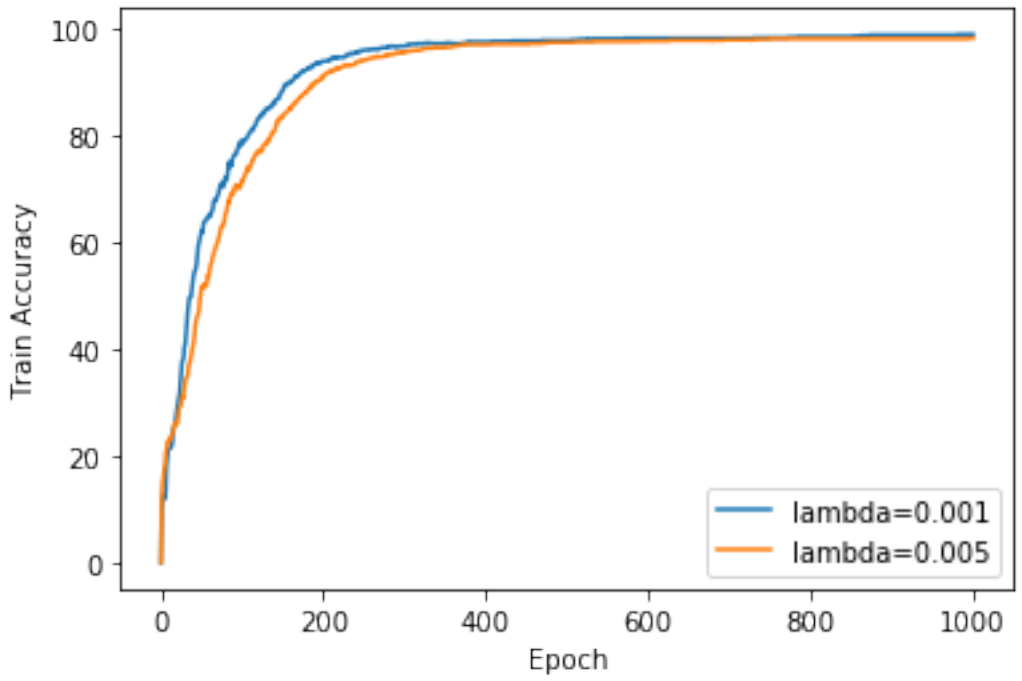
```
train acc: 10.100000000000001    test acc: 9.2   at epoch: 0
train acc: 79.2  test acc: 62.64999999999999     at epoch: 100
train acc: 93.89999999999999     test acc: 75.35         at epoch: 200
```

```
train acc: 96.7  test acc: 77.4  at epoch: 300
train acc: 97.5  test acc: 77.85         at epoch: 400
train acc: 97.8  test acc: 78.5  at epoch: 500
train acc: 98.2  test acc: 78.64999999999999     at epoch: 600
train acc: 98.2  test acc: 78.7  at epoch: 700
train acc: 98.5  test acc: 78.85         at epoch: 800
train acc: 98.8  test acc: 79.05         at epoch: 900
```

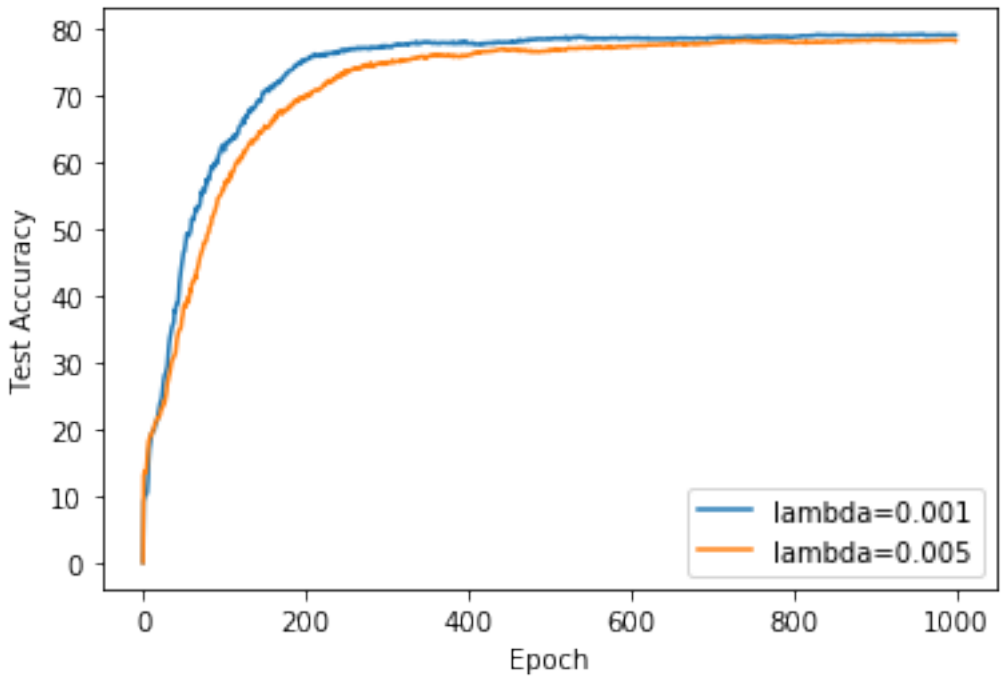
[113]:
```
loss = torch.nn.CrossEntropyLoss()
l1_005_metrics, l1_005_model = train(trainloader, train_data, train_labels,␣
↪test_data, test_labels,
     num_epochs, num_input, num_classes, learning_rate, loss,␣
↪regularization="l1", reg_lambda=0.005)
```

```
train acc: 11.899999999999999    test acc: 13.100000000000001    at epoch: 0
train acc: 71.8  test acc: 56.99999999999999     at epoch: 100
train acc: 91.3  test acc: 70.1  at epoch: 200
train acc: 95.6  test acc: 75.0  at epoch: 300
train acc: 97.0  test acc: 76.1  at epoch: 400
train acc: 97.39999999999999     test acc: 76.7  at epoch: 500
train acc: 97.7  test acc: 77.45         at epoch: 600
train acc: 97.89999999999999     test acc: 77.9  at epoch: 700
train acc: 98.1  test acc: 78.0  at epoch: 800
train acc: 98.1  test acc: 78.35         at epoch: 900
```

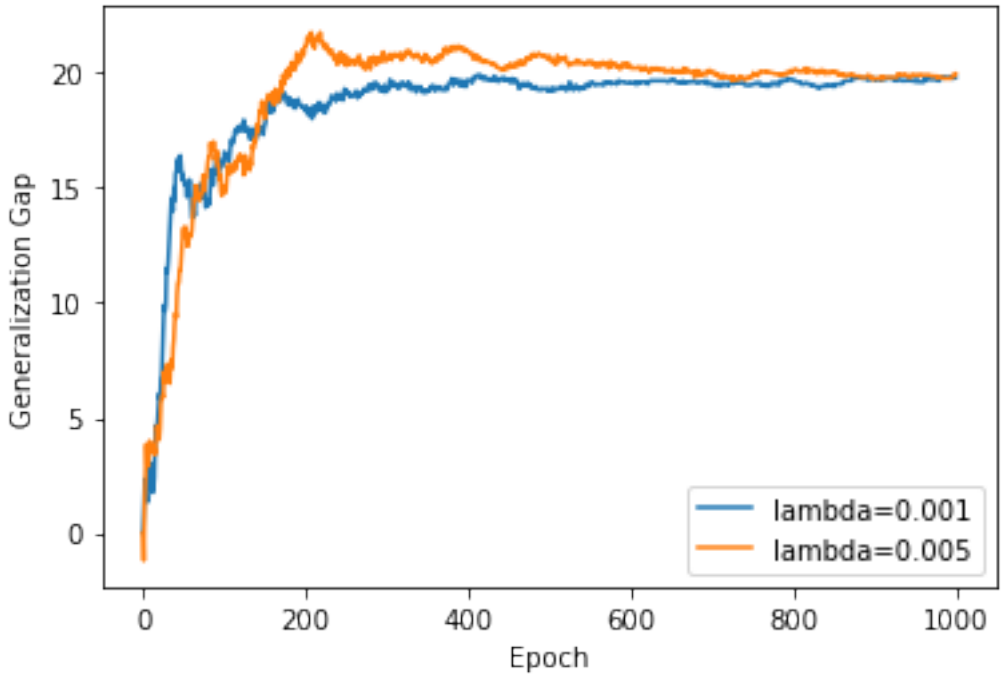
[122]:
```
plt.plot(l1_001_metrics[:,0], label="lambda=0.001")
plt.plot(l1_005_metrics[:,0], label="lambda=0.005")
plt.xlabel("Epoch")
plt.ylabel("Train Accuracy")
plt.legend()
plt.show()
```



[123]:
```
plt.plot(l1_001_metrics[:,1], label="lambda=0.001")
plt.plot(l1_005_metrics[:,1], label="lambda=0.005")
plt.xlabel("Epoch")
plt.ylabel("Test Accuracy")
plt.legend()
plt.show()
```



[124]:
```
plt.plot(l1_001_metrics[:,0]-l1_001_metrics[:,1], label="lambda=0.001")
plt.plot(l1_005_metrics[:,0]-l1_005_metrics[:,1], label="lambda=0.005")
plt.xlabel("Epoch")
plt.ylabel("Generalization Gap")
plt.legend()
plt.show()
```



[ ]:

[ ]:

1.5.2 L2 Regularization

[114]:
```
from prob4 import *
```

[115]:
```
loss = torch.nn.CrossEntropyLoss()
l2_001_metrics, l2_001_model = train(trainloader, train_data, train_labels,␣
↪test_data, test_labels,
     num_epochs, num_input, num_classes, learning_rate, loss,␣
↪regularization="l2", reg_lambda=0.001)
```

```
train acc: 10.0  test acc: 9.5   at epoch: 0
train acc: 71.0  test acc: 55.35         at epoch: 100
train acc: 91.8  test acc: 70.6  at epoch: 200
train acc: 95.6  test acc: 74.45         at epoch: 300
train acc: 96.8  test acc: 75.64999999999999     at epoch: 400
train acc: 97.8  test acc: 76.55         at epoch: 500
train acc: 98.5  test acc: 76.8  at epoch: 600
train acc: 98.6  test acc: 77.0  at epoch: 700
train acc: 98.8  test acc: 77.60000000000001     at epoch: 800
train acc: 98.9  test acc: 77.75         at epoch: 900
```

[117]:
```
loss = torch.nn.CrossEntropyLoss()
l2_01_metrics, l2_01_model = train(trainloader, train_data, train_labels,␣
↪test_data, test_labels,
     num_epochs, num_input, num_classes, learning_rate, loss,␣
↪regularization="l2", reg_lambda=0.01)
```

```
train acc: 10.9  test acc: 10.75         at epoch: 0
train acc: 69.1  test acc: 54.6  at epoch: 100
train acc: 89.60000000000001     test acc: 69.6  at epoch: 200
train acc: 95.39999999999999     test acc: 74.9  at epoch: 300
train acc: 96.6  test acc: 75.8  at epoch: 400
train acc: 97.1  test acc: 77.05         at epoch: 500
train acc: 97.5  test acc: 77.35         at epoch: 600
train acc: 97.8  test acc: 77.7  at epoch: 700
train acc: 97.8  test acc: 78.2  at epoch: 800
train acc: 98.2  test acc: 78.10000000000001     at epoch: 900
```

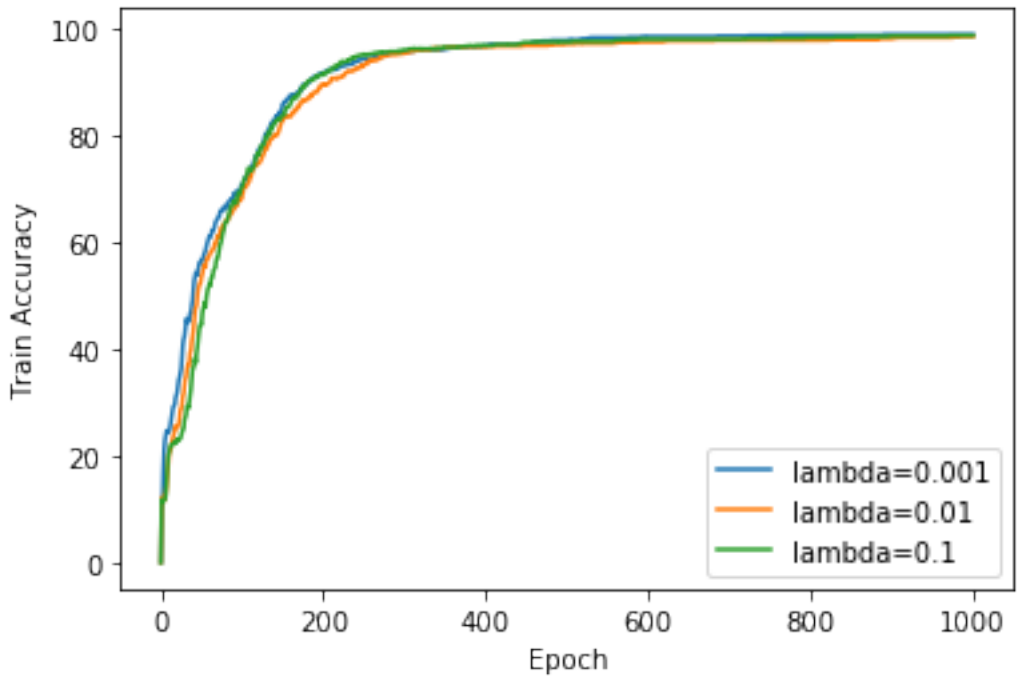
[118]:
```
loss = torch.nn.CrossEntropyLoss()
l2_1_metrics, l2_1_model = train(trainloader, train_data, train_labels,␣
↪test_data, test_labels,
     num_epochs, num_input, num_classes, learning_rate, loss,␣
↪regularization="l2", reg_lambda=0.1)
```

```
train acc: 11.799999999999999    test acc: 11.85         at epoch: 0
train acc: 70.7  test acc: 52.5  at epoch: 100
train acc: 91.60000000000001     test acc: 71.95         at epoch: 200
train acc: 96.0  test acc: 76.05         at epoch: 300
train acc: 96.8  test acc: 76.8  at epoch: 400
train acc: 97.5  test acc: 77.0  at epoch: 500
train acc: 98.0  test acc: 77.0  at epoch: 600
train acc: 98.0  test acc: 76.85         at epoch: 700
train acc: 98.3  test acc: 77.55         at epoch: 800
train acc: 98.6  test acc: 77.85         at epoch: 900
```

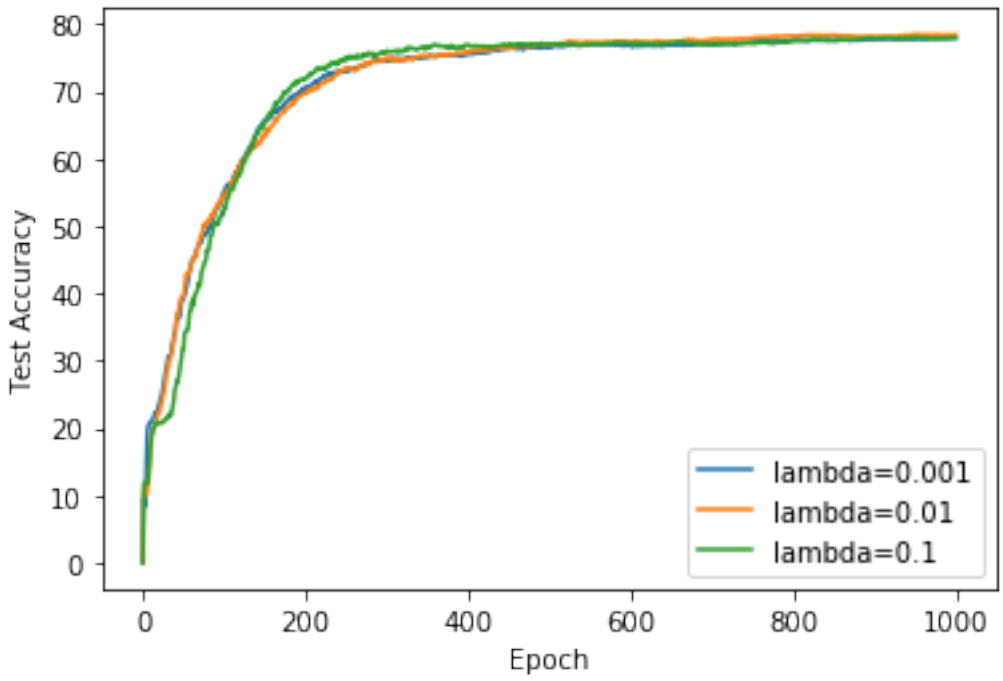
[ ]:
```

```

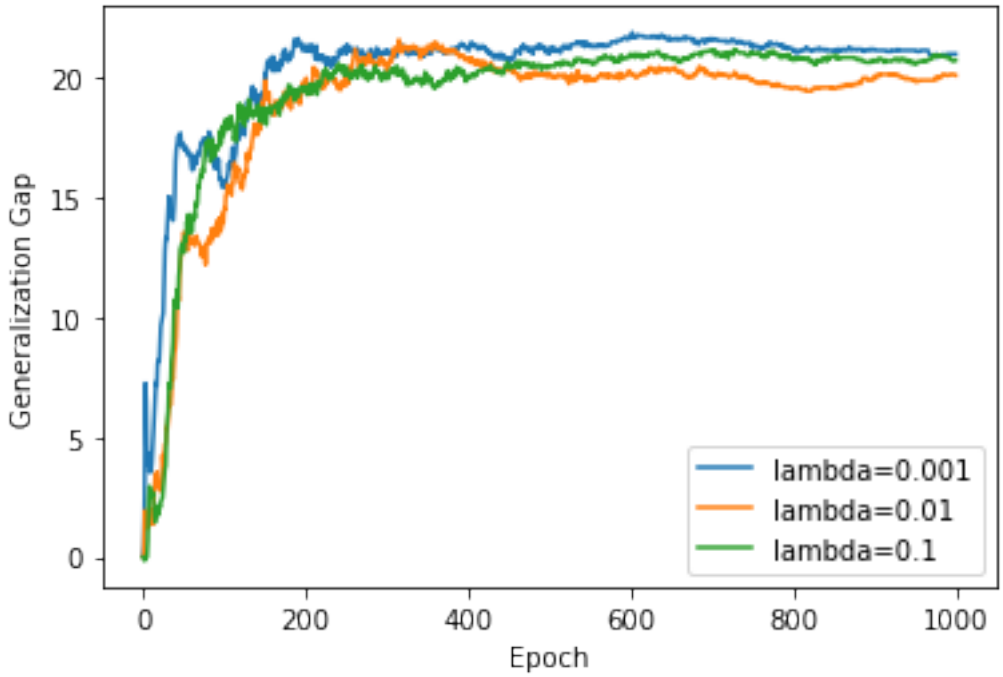
[126]:
```
plt.plot(l2_001_metrics[:,0], label="lambda=0.001")
plt.plot(l2_01_metrics[:,0], label="lambda=0.01")
plt.plot(l2_1_metrics[:,0], label="lambda=0.1")
plt.xlabel("Epoch")
plt.ylabel("Train Accuracy")
plt.legend()
plt.show()
```



[127]:
```
plt.plot(l2_001_metrics[:,1], label="lambda=0.001")
plt.plot(l2_01_metrics[:,1], label="lambda=0.01")
plt.plot(l2_1_metrics[:,1], label="lambda=0.1")
plt.xlabel("Epoch")
plt.ylabel("Test Accuracy")
plt.legend()
plt.show()
```



[125]:
```
plt.plot(l2_001_metrics[:,0]-l2_001_metrics[:,1], label="lambda=0.001")
plt.plot(l2_01_metrics[:,0]-l2_01_metrics[:,1], label="lambda=0.01")
plt.plot(l2_1_metrics[:,0]-l2_1_metrics[:,1], label="lambda=0.1")
plt.xlabel("Epoch")
plt.ylabel("Generalization Gap")
plt.legend()
plt.show()
```



[ ]:

[ ]:

1.5.3 Dropout

[129]:
```
loss = torch.nn.CrossEntropyLoss()
dropout_05_metrics, dropout_05_model = train(trainloader, train_data,␣
 ↪train_labels, test_data, test_labels,
      num_epochs, num_input, num_classes, learning_rate, loss, dropout_p=0.05)
```

```
train acc: 11.0  test acc: 10.549999999999999    at epoch: 0
train acc: 69.5  test acc: 54.75         at epoch: 100
train acc: 90.8  test acc: 70.8  at epoch: 200
train acc: 95.19999999999999     test acc: 73.8  at epoch: 300
train acc: 96.2  test acc: 75.55         at epoch: 400
train acc: 96.8  test acc: 77.60000000000001     at epoch: 500
train acc: 97.2  test acc: 77.2  at epoch: 600
train acc: 97.89999999999999     test acc: 77.8  at epoch: 700
train acc: 97.7  test acc: 77.14999999999999     at epoch: 800
train acc: 98.1  test acc: 77.5  at epoch: 900
```

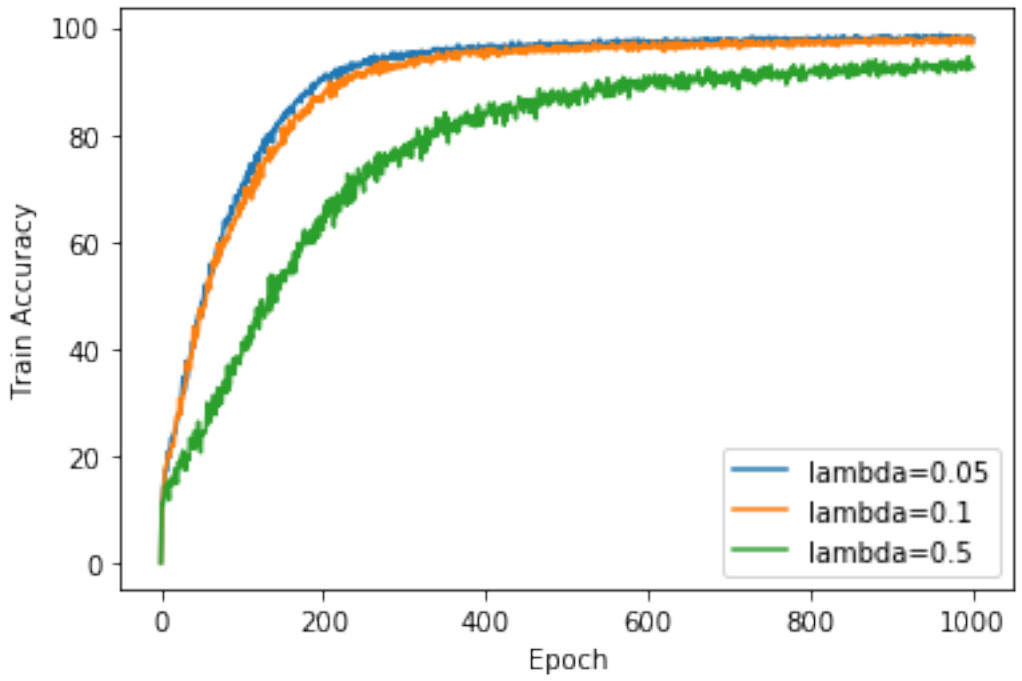
[130]:
```
loss = torch.nn.CrossEntropyLoss()
dropout_1_metrics, dropout_1_model = train(trainloader, train_data,␣
↪train_labels, test_data, test_labels,
      num_epochs, num_input, num_classes, learning_rate, loss, dropout_p=0.1)
```

```
train acc: 10.0  test acc: 8.85  at epoch: 0
train acc: 66.4  test acc: 52.1  at epoch: 100
train acc: 87.7  test acc: 67.30000000000001     at epoch: 200
train acc: 93.30000000000001     test acc: 72.65         at epoch: 300
train acc: 96.1  test acc: 74.45         at epoch: 400
train acc: 96.2  test acc: 75.05         at epoch: 500
train acc: 96.1  test acc: 76.6  at epoch: 600
train acc: 97.1  test acc: 75.94999999999999     at epoch: 700
train acc: 97.0  test acc: 76.1  at epoch: 800
train acc: 97.3  test acc: 77.64999999999999     at epoch: 900
```

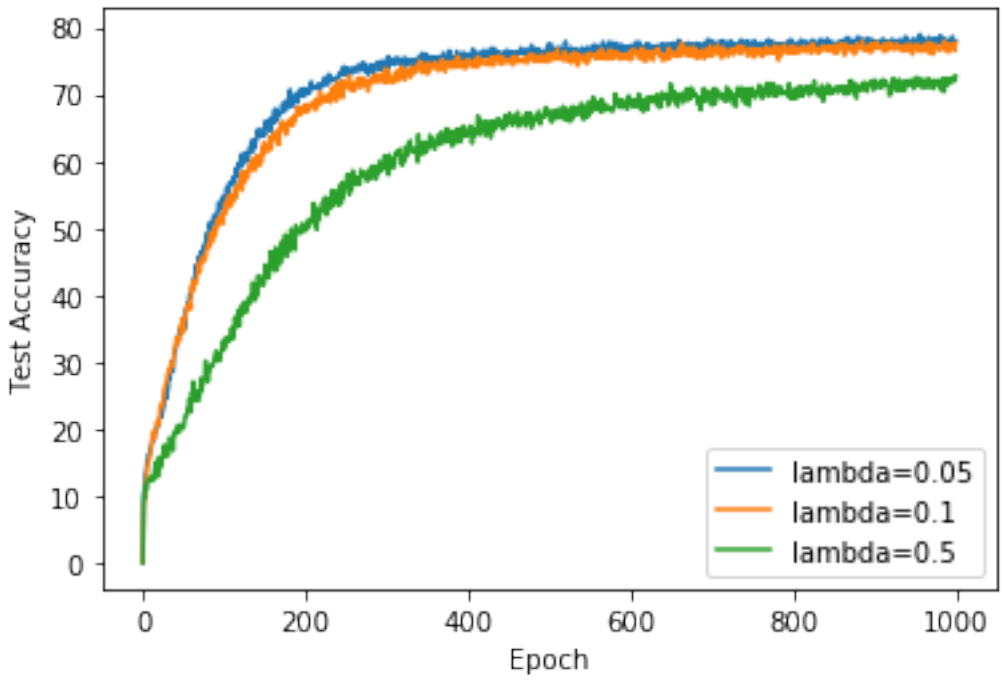
[131]:
```
loss = torch.nn.CrossEntropyLoss()
dropout_5_metrics, dropout_5_model = train(trainloader, train_data,␣
↪train_labels, test_data, test_labels,
      num_epochs, num_input, num_classes, learning_rate, loss, dropout_p=0.5)
```

```
train acc: 10.0  test acc: 9.9   at epoch: 0
train acc: 40.699999999999996    test acc: 32.4  at epoch: 100
train acc: 66.4  test acc: 49.95         at epoch: 200
train acc: 76.3  test acc: 60.650000000000006    at epoch: 300
train acc: 83.5  test acc: 65.0  at epoch: 400
train acc: 87.5  test acc: 66.7  at epoch: 500
train acc: 90.9  test acc: 68.95         at epoch: 600
train acc: 91.7  test acc: 68.89999999999999     at epoch: 700
train acc: 91.5  test acc: 69.35         at epoch: 800
train acc: 91.3  test acc: 71.2  at epoch: 900
```

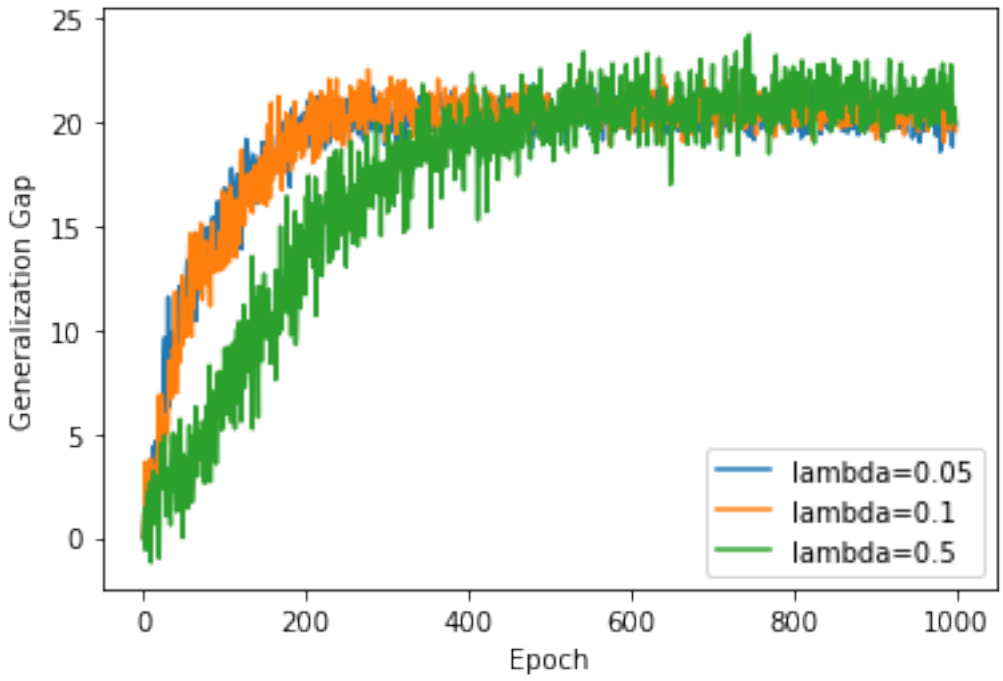
[133]:
```
plt.plot(dropout_05_metrics[:,0], label="lambda=0.05")
plt.plot(dropout_1_metrics[:,0], label="lambda=0.1")
plt.plot(dropout_5_metrics[:,0], label="lambda=0.5")
plt.xlabel("Epoch")
plt.ylabel("Train Accuracy")
plt.legend()
plt.show()
```



[134]:
```
plt.plot(dropout_05_metrics[:,1], label="lambda=0.05")
plt.plot(dropout_1_metrics[:,1], label="lambda=0.1")
plt.plot(dropout_5_metrics[:,1], label="lambda=0.5")
plt.xlabel("Epoch")
plt.ylabel("Test Accuracy")
plt.legend()
plt.show()
```



[136]:
```
plt.plot(dropout_05_metrics[:,0]-dropout_05_metrics[:,1], label="lambda=0.05")
plt.plot(dropout_1_metrics[:,0]-dropout_1_metrics[:,1], label="lambda=0.1")
plt.plot(dropout_5_metrics[:,0]-dropout_5_metrics[:,1], label="lambda=0.5")
plt.xlabel("Epoch")
plt.ylabel("Generalization Gap")
plt.legend()
plt.show()
```



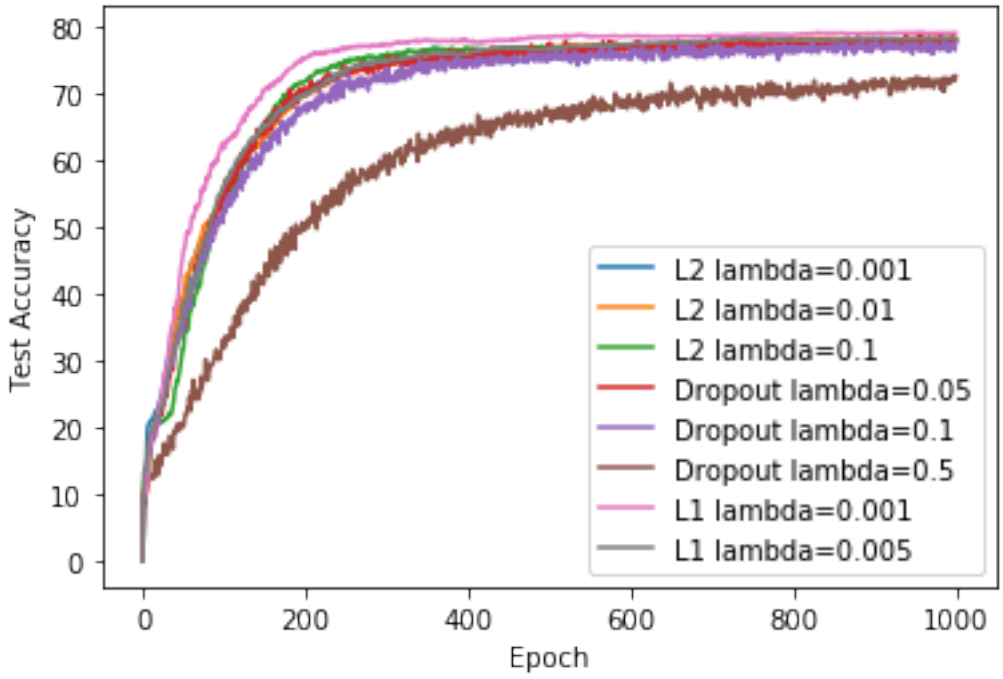
1.5.4 Plot Loss Data

[137]:
```
plt.plot(l2_001_metrics[:,1], label="L2 lambda=0.001")
plt.plot(l2_01_metrics[:,1], label="L2 lambda=0.01")
plt.plot(l2_1_metrics[:,1], label="L2 lambda=0.1")

plt.plot(dropout_05_metrics[:,1], label="Dropout lambda=0.05")
plt.plot(dropout_1_metrics[:,1], label="Dropout lambda=0.1")
plt.plot(dropout_5_metrics[:,1], label="Dropout lambda=0.5")

plt.plot(l1_001_metrics[:,1], label="L1 lambda=0.001")
plt.plot(l1_005_metrics[:,1], label="L1 lambda=0.005")

plt.xlabel("Epoch")
plt.ylabel("Test Accuracy")
plt.legend()
plt.show()
```



Generally speaking, there does seem to be sensitivity to the regularization methods, that either increase or decrease the accuracy beyond the unregularized values. Specifically, L2 regularization seems to have little effect, regardless of the lambda chosen. This may be due to the fact that small weights have a relatively minimal impact with such a small network. However, sparsifying the model by dropping neurons does seem to help, which suggests that the model may be overparameterized with too many neurons. Finally, we validate our results by showing that if 50% of our nodes are dropped out, then we have the worst accuracy of all our models. This suggests that at least half our neurons are learning useful information for classification.

[ ]: