From-scratch Network

May 2, 2022

1 Deep Learning From-scratch Network Training

Zach Kangas

5/2/22

Dr. Yoder

Deep Learning

2 Introduction

In this report, we look at using the from-scratch neural network created with PyTorch to train on a linear toy dataset, then the CIFAR-10 and MNIST flattened datasets. The results from the training of each dataset is described after training, in a final validation statistic and a graph of the loss and accuracy of the training and validation sets.

2.1 Importing Relevant Libraries

[1]:
```
import torch
import numpy as np
import matplotlib.pyplot as plt
import torchvision
import warnings
import os.path
import math
import seaborn as sns

import NeuralNetwork
import Client
```

2.2 Setting up dtype, device, and datapaths

[2]:
```
# warnings.filterwarnings('ignore')  # If you see warnings that you know you␣
↪can ignore, it can be useful to enable this.

# For fashion-MNIST and similar problems
```

```
DATA_ROOT = '/data/cs3450/data/'
FASHION_MNIST_TRAINING = '/data/cs3450/data/fashion_mnist_flattened_training.
↪npz'
FASHION_MNIST_TESTING = '/data/cs3450/data/fashion_mnist_flattened_testing.npz'
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'

DTYPE = torch.float32
# 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
```

2.3 Importing training data

[3]:
```
def create_linear_training_data(training_points):
    """
    This method simply rotates points in a 2D space.
    Be sure to use L2 regression in the place of the final softmax layer before␣
 ↪testing on this
    data!
    :return: (x,y) the dataset. x is a numpy array where columns are training␣
 ↪samples and
             y is a numpy array where columns are one-hot labels for the␣
 ↪training sample.
    """
    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

def create_folded_training_data():
    """
    This method introduces a single non-linear fold into the sort of data␣
 ↪created by create_linear_training_data. Be sure to REMOVE the final softmax␣
 ↪layer before testing on this data!
    Be sure to use L2 regression in the place of the final softmax layer before␣
 ↪testing on this
```

```
    data!
    :return: (x,y) the dataset. x is a numpy array where columns are training␣
↪samples and
             y is a numpy array where columns are one-hot labels for the␣
↪training sample.
    """
    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


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 points␣
↪and insideness is true if the point is inside the square.
    """
    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␣
↪origin
    insideness = (normal[:,:,np.newaxis] * vectors).sum(axis=0)
    insideness = insideness.T
    insideness = insideness > 0
    insideness = insideness.all(axis=1)
    return points, insideness


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

```python
    :return: (points, insideness) the dataset. points is a 2xN array of points and insideness is true if the point is inside the square.
    """
    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


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 unneeded downloads.
    :return: (x,y) the dataset. x is a numpy array where columns are training samples and
             y is a numpy array where columns are one-hot labels for the training sample.
    """
    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 train else 'testing')
        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 to a numpy array"""

            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=download)
        elif dataset == 'CIFAR-10':
            data = torchvision.datasets.CIFAR10(
                root=DATA_ROOT, train=train, transform=ToTorch(), download=download)
        elif dataset == 'CIFAR-100':
            data = torchvision.datasets.CIFAR100(
                root=DATA_ROOT, train=train, transform=ToTorch(), download=download)
        else:
            raise ValueError('This code should be unreachable because of a previous check.')
        x = torch.zeros((len(data[0][0].flatten()), len(data)),dtype=torch.float32)
        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.detach().numpy(), y=y.detach().numpy())
    return x, y
```

3 Training the Network - Linear

In this section, I use a small network to predict on the linear dataset. This really should not need any more than 3-5 nodes in the hidden layer, but I chose 10 for this example as the train time was extremely quick regardless.

[4]:
```
x, y = create_linear_training_data(10000)
epochs = 10

nn = NeuralNetwork.Network(x.shape[0], y.shape[0], dtype=torch.float32,␣
 ↪loss="l2", regularization_factor=0.01, learning_rate=0.001)
lin1 = nn.add_linear_generated(num_nodes=10, w=0.05, wo=0, b=0, bo=0,␣
 ↪regularization=True)
rel1 = nn.add_relu()
lin2 = nn.add_linear_generated(num_nodes=2, w=0.05, wo=0, b=0, bo=0,␣
 ↪regularization=True)


client = Client.Client(nn, x, y)
tloss, tacc = client.train_stats()
vloss, vacc = client.validation_stats()
print("E: -1", "\ttL:", tloss, "\ttA:", tacc, "\tvL:", vloss, "\tvA:", vacc)
train_data = client.train(epochs, 1, verbose=True)
```

```
E: -1   tL: 1.9948890209197998  tA: 0.4767500162124634  vL: 1.882861614227295
vA: 0.5045000314712524
E: 0    tL: 0.0788913443684578  tA: 0.9827500581741333  vL: 0.06811783462762833
vA: 0.9775000214576721
E: 1    tL: 0.022522913292050 36        tA: 0.9888750314712524  vL:
0.019315805286169052        vA: 0.9900000691413879
E: 2    tL: 0.013064330443739891        tA: 0.9937500357627869  vL:
0.010838055051863194        vA: 0.9915000200271606
E: 3    tL: 0.008565248921513557        tA: 0.9961250424385071  vL:
0.007008089683949947        vA: 0.9965000748634338
E: 4    tL: 0.006365333218127489        tA: 0.9978750348091125  vL:
0.005219032522290945        vA: 0.9985000491142273
E: 5    tL: 0.005125655326992273        tA: 0.9982500672340393  vL:
0.004231084138154983 5        vA: 0.9985000491142273
E: 6    tL: 0.004361961502581835        tA: 0.9986250400543213  vL:
0.0036184692289680243        vA: 0.9985000491142273
E: 7    tL: 0.003849930362775922        tA: 0.9987500309944153  vL:
0.0031996257603168488        vA: 0.9985000491142273
E: 8    tL: 0.003465912537649274        tA: 0.9987500309944153  vL:
0.0028836484998464584        vA: 0.9985000491142273
E: 9    tL: 0.0031606603879481554        tA: 0.9988750219345093  vL:
0.0026310631074011326        vA: 0.9985000491142273
```

3.1 Statistics

Final Training Loss*: 0.003

Final Training Accuracy*: 0.999

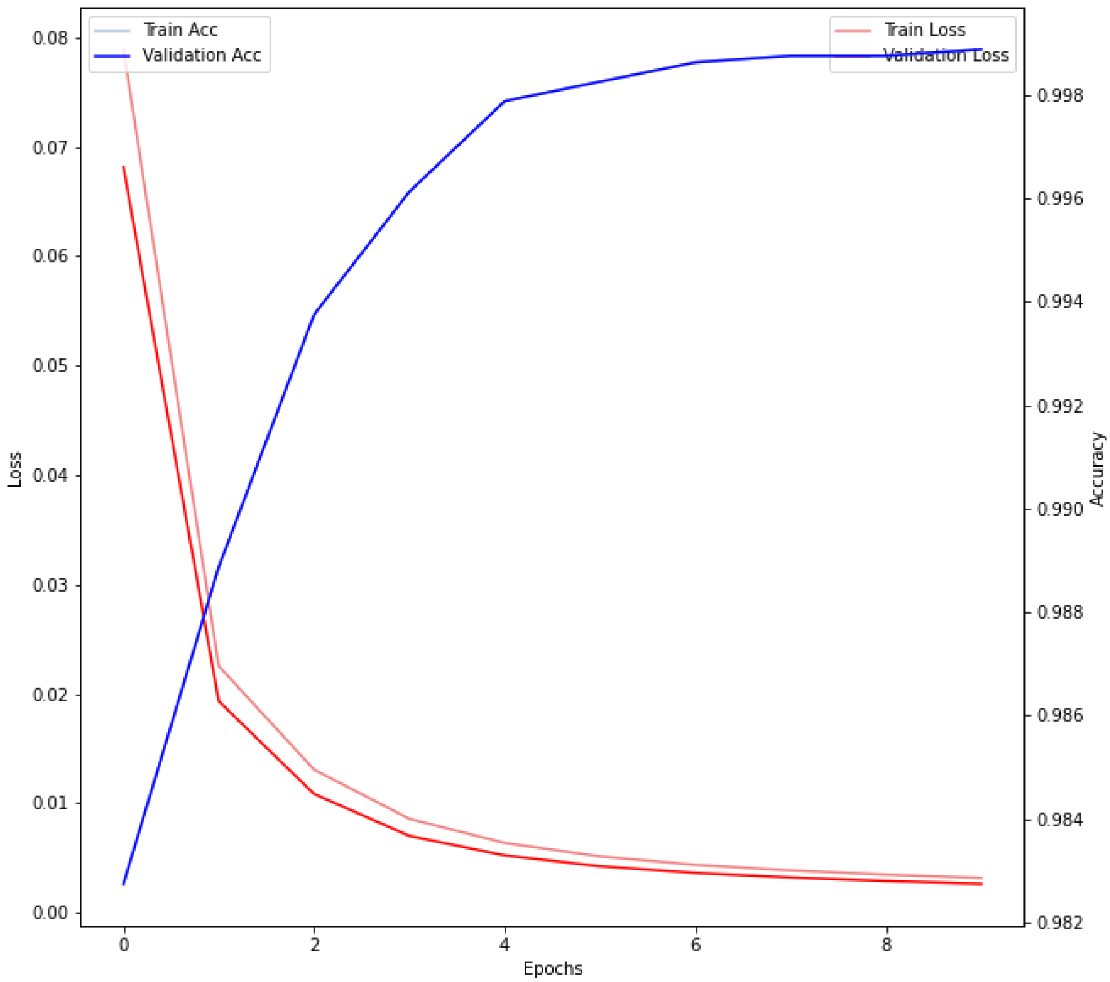
Final Validation Loss: 0.003

Final Validation Accuracy: 0.999

*: Validation and training are basically identical

3.1.1 Training Graph

[5]:
```
fig = plt.figure(figsize=(10,10))
pls = np.array(train_data)
ax = sns.lineplot(x=pls[:,0], y=pls[:,1], color="lightcoral", label="Train␣
 ↪Loss")
ax = sns.lineplot(x=pls[:,0], y=pls[:,3], color="r", label="Validation Loss")
ax.set_xlabel("Epochs")
ax.set_ylabel("Loss")
ax2 = plt.twinx()
ax2.set_ylabel("Accuracy")
sns.lineplot(x=pls[:,0], y=pls[:,2], color="lightsteelblue", label="Train Acc")
sns.lineplot(x=pls[:,0], y=pls[:,2], color="b", label="Validation Acc")
plt.show()
```



4 Training the Network - MNIST

[6]:
```
dataset = 'Fashion-MNIST'

x, y = load_dataset_flattened(train=True, dataset=dataset, download=False)
epochs = 50

nn = NeuralNetwork.Network(x.shape[0], y.shape[0], loss="ce",␣
 ↪regularization_factor=0.00001, learning_rate=0.001)
lin3 = nn.add_linear_generated(num_nodes=800, w=0.001, wo=0, b=0, bo=0,␣
 ↪regularization=True)
rel = nn.add_relu()
lin3 = nn.add_linear_generated(num_nodes=300, w=0.0025, wo=0, b=0, bo=0,␣
 ↪regularization=True)
rel = nn.add_relu()
```

```
lin4 = nn.add_linear_generated(num_nodes=y.shape[0], w=0.03, wo=0, b=0, bo=0,␣
 ↪regularization=True)
nn.add_softmax()

client = Client.Client(nn, x, y)
tloss, tacc = client.train_stats()
vloss, vacc = client.validation_stats()
print("E: -1", "\ttL:", tloss, "\ttA:", tacc, "\tvL:", vloss, "\tvA:", vacc)
train_data = client.train(epochs, 1, verbose=True)
```

```
Loading cached flattened data for Fashion-MNIST training
E: -1   tL: 2.301969289779663   tA: 0.10106249898672104         vL:
2.3022379875183105  vA: 0.09574999660253525
E: 0    tL: 1.0199207067489624  tA: 0.6098541617393494  vL: 1.0165126323699951
vA: 0.6106666326522827
E: 1    tL: 0.7970491647720337  tA: 0.7301666736602783  vL: 0.8024205565452576
vA: 0.7242500185966492
E: 2    tL: 0.6782697439193726  tA: 0.7927708029747009  vL: 0.6892938613891602
vA: 0.7858332991600037
E: 3    tL: 0.5547126531600952  tA: 0.835979163646698   vL: 0.5694445371627808
vA: 0.828249990940094
E: 4    tL: 0.5016655921936035  tA: 0.8491041660308838  vL: 0.520009458065033
vA: 0.8395833373069763
E: 5    tL: 0.4686982035636902  tA: 0.8564791679382324  vL: 0.49052560329437256
vA: 0.8460000157356262
E: 6    tL: 0.4433543384075165  tA: 0.8630833029747009  vL: 0.46792879700660706
vA: 0.8528333306312561
E: 7    tL: 0.42368823289871216         tA: 0.867354154586792   vL:
0.4514448046684265  vA: 0.856249988079071
E: 8    tL: 0.4073924124240875  tA: 0.871749997138977   vL: 0.43798235058784485
vA: 0.859749972820282
E: 9    tL: 0.3929104804992676  tA: 0.8749791383743286  vL: 0.4260786175727844
vA: 0.8619999885559082
E: 10   tL: 0.3799373209476471  tA: 0.8788958191871643  vL: 0.41609740257263184
vA: 0.8644166588783264
E: 11   tL: 0.3679186701774597  tA: 0.883104145526886   vL: 0.40698710083961487
vA: 0.8667500019073486
E: 12   tL: 0.3589191138744354  tA: 0.8858749866485596  vL: 0.40038353204727173
vA: 0.8667500019073486
E: 13   tL: 0.3506874144077301  tA: 0.8873957991600037  vL: 0.3951770067214966
vA: 0.8680832982063293
E: 14   tL: 0.3408052623271942  tA: 0.890625    vL: 0.38824355602264404
vA: 0.8704166412353516
E: 15   tL: 0.33359503746032715         tA: 0.8930000066757202  vL:
0.3835214078426361  vA: 0.8709999918937683
E: 16   tL: 0.32471516728401184         tA: 0.895145833492279   vL:
0.3775201737880707  vA: 0.8725833296775818
```

E: 17 tL: 0.3182857632637024 tA: 0.8966875076293945 vL: 0.3744330108165741
vA: 0.8734166622161865
E: 18 tL: 0.3099108934402466 tA: 0.8997499942779541 vL: 0.3686022162437439
vA: 0.8758333325386047
E: 19 tL: 0.30336180329322815 tA: 0.9010416865348816 vL:
0.36507558822631836 vA: 0.8761666417121887
E: 20 tL: 0.29601964354515076 tA: 0.9032708406448364 vL:
0.360989511013031 vA: 0.8775833249092102
E: 21 tL: 0.2914585769176483 tA: 0.9045208096504211 vL: 0.3592927157878876
vA: 0.878083348274231
E: 22 tL: 0.2851117253303528 tA: 0.9055416584014893 vL: 0.3562021553516388
vA: 0.8774999976158142
E: 23 tL: 0.2775116264820099 tA: 0.909333348274231 vL: 0.35242822766304016
vA: 0.8779166340827942
E: 24 tL: 0.27286866307258606 tA: 0.9104791283607483 vL:
0.3509041368961334 vA: 0.8787499666213989
E: 25 tL: 0.2669496536254883 tA: 0.9120833277702332 vL: 0.3480924963951111
vA: 0.8808333277702332
E: 26 tL: 0.2623220384120941 tA: 0.9133749604225159 vL: 0.3467816114425659
vA: 0.8814166784286499
E: 27 tL: 0.2570047080516815 tA: 0.914354145526886 vL: 0.3445451259613037
vA: 0.8798333406448364
E: 28 tL: 0.25267696380615234 tA: 0.9165208339691162 vL:
0.3447311222553253 vA: 0.8799166679382324
E: 29 tL: 0.24646137654781342 tA: 0.9169583320617676 vL:
0.3419124186038971 vA: 0.8807500004768372
E: 30 tL: 0.24111855030059814 tA: 0.9186874628067017 vL:
0.33942484855651855 vA: 0.8819166421890259
E: 31 tL: 0.24070051312446594 tA: 0.918999969959259 vL:
0.3421674072742462 vA: 0.8805000185966492
E: 32 tL: 0.23306208848953247 tA: 0.9228541851043701 vL:
0.337630033493042 vA: 0.8828333020210266
E: 33 tL: 0.2310403287410736 tA: 0.9213333129882812 vL: 0.3385462164878845
vA: 0.8816666603088379
E: 34 tL: 0.22413130104541779 tA: 0.9251874685287476 vL:
0.3358246684074402 vA: 0.8844999670982361
E: 35 tL: 0.22544534504413605 tA: 0.9226458072662354 vL:
0.3413766622543335 vA: 0.8818333148956299
E: 36 tL: 0.21809890866279602 tA: 0.9243957996368408 vL:
0.33726605772972107 vA: 0.8799999952316284
E: 37 tL: 0.21374984085559845 tA: 0.9277708530426025 vL:
0.33766958117485046 vA: 0.8824166655540466
E: 38 tL: 0.2118811011314392 tA: 0.9261249899864197 vL: 0.34009861946105957
vA: 0.8815833330154419
E: 39 tL: 0.2011374831199646 tA: 0.929562509059906 vL: 0.3334323763847351
vA: 0.8803333044052124
E: 40 tL: 0.19780659675598145 tA: 0.9322291612625122 vL:
0.33363547921180725 vA: 0.8814166784286499

```
E: 41   tL: 0.19463203847408295         tA: 0.9322708249092102  vL:
0.33499154448509216         vA: 0.8822500109672546
E: 42   tL: 0.1957651823759079  tA: 0.9289374947547913  vL: 0.3398110568523407
vA: 0.8805833458900452
E: 43   tL: 0.2429502159357071  tA: 0.9192500114440918  vL: 0.34988078474998474
vA: 0.8775833249092102
E: 44   tL: 0.21054844558238983         tA: 0.9306041598320007  vL:
0.3362560272216797  vA: 0.8819999694824219
E: 45   tL: 0.19667980074882507         tA: 0.9354791641235352  vL:
0.33261504769325256         vA: 0.8823333382606506
E: 46   tL: 0.19415238499641418         tA: 0.9334166646003723  vL:
0.3360462486743927  vA: 0.8822500109672546
E: 47   tL: 0.1902884840965271  tA: 0.9346041679382324  vL: 0.33751821517944336
vA: 0.8814166784286499
E: 48   tL: 0.18186955153942108         tA: 0.937874972820282   vL:
0.3320619761943817  vA: 0.8849999904632568
E: 49   tL: 0.18418346345424652         tA: 0.9369791746139526  vL:
0.3405294418334961  vA: 0.8833333253860474
```

4.1 Statistics

Final Training Loss: 0.184

Final Training Accuracy: 0.937

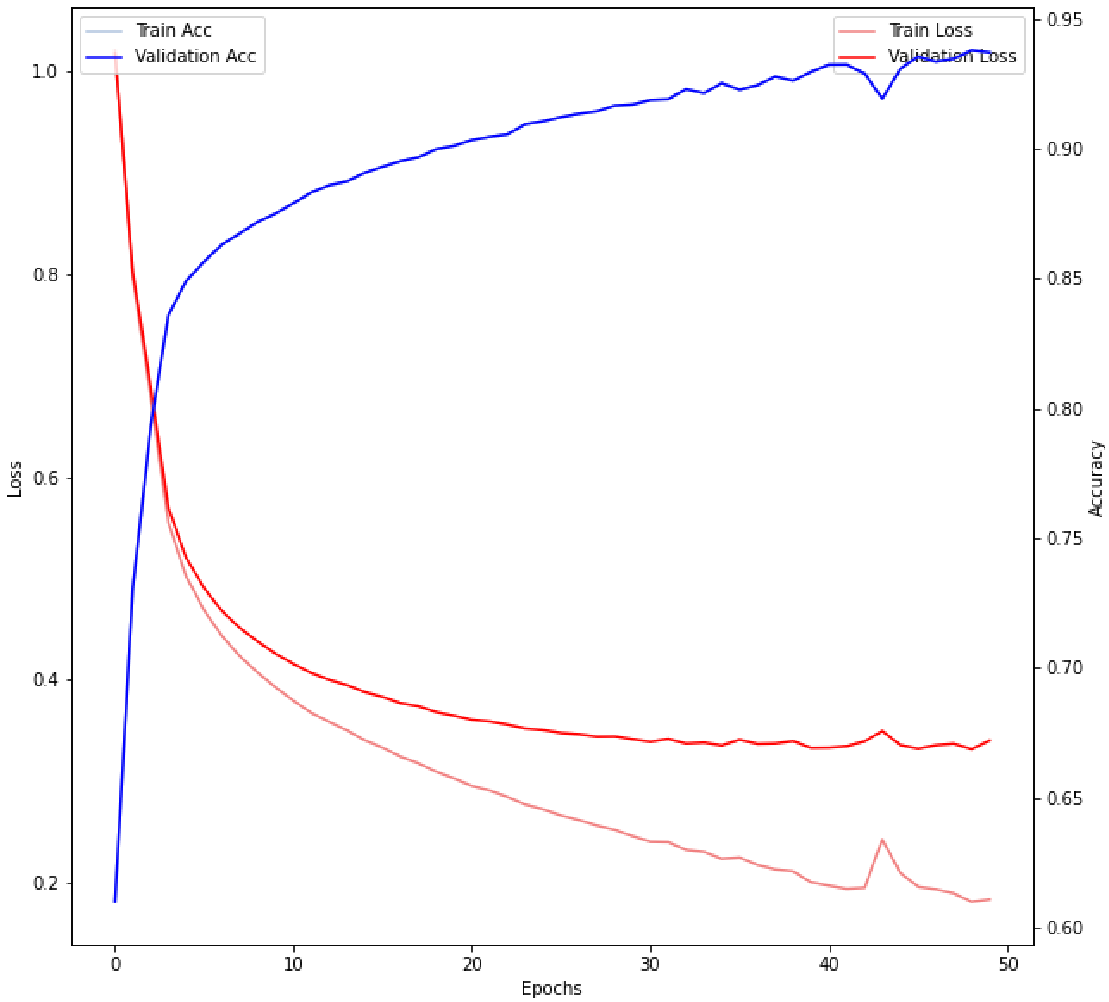
Final Validation Loss: 0.341

Final Validation Accuracy: 0.883

4.1.1 Training Graph

[7]:

```
fig = plt.figure(figsize=(10,10))
pls = np.array(train_data)
ax = sns.lineplot(x=pls[:,0], y=pls[:,1], color="lightcoral", label="Train␣
↪Loss")
ax = sns.lineplot(x=pls[:,0], y=pls[:,3], color="r", label="Validation Loss")
ax.set_xlabel("Epochs")
ax.set_ylabel("Loss")
ax2 = plt.twinx()
ax2.set_ylabel("Accuracy")
sns.lineplot(x=pls[:,0], y=pls[:,2], color="lightsteelblue", label="Train Acc")
sns.lineplot(x=pls[:,0], y=pls[:,2], color="b", label="Validation Acc")
plt.show()
```



5 Training the Network - CIFAR-10

[8]:
```
%%time
dataset = 'CIFAR-10'

x, y = load_dataset_flattened(train=True, dataset=dataset, download=False)
epochs = 50

nn = NeuralNetwork.Network(x.shape[0], y.shape[0], loss="ce",␣
 ↪regularization_factor=0.0001, learning_rate=0.0002)
lin3 = nn.add_linear_generated(num_nodes=800, w=0.001, wo=0, b=0, bo=0,␣
 ↪regularization=True)
rel = nn.add_relu()
lin3 = nn.add_linear_generated(num_nodes=300, w=0.0025, wo=0, b=0, bo=0,␣
 ↪regularization=True)
```

```
rel = nn.add_relu()
lin4 = nn.add_linear_generated(num_nodes=y.shape[0], w=0.03, wo=0, b=0, bo=0,␣
 ↪regularization=True)
nn.add_softmax()

client = Client.Client(nn, x, y)

tloss, tacc = client.train_stats()
vloss, vacc = client.validation_stats()
print("E: -1", "\ttL:", tloss, "\ttA:", tacc, "\tvL:", vloss, "\tvA:", vacc)
train_data = client.train(epochs, 1, verbose=True)
```

```
Loading cached flattened data for CIFAR-10 training
E: -1   tL: 2.306262493133545   tA: 0.10119999945163727         vL:
2.306717872619629   vA: 0.09519999474287033
E: 0    tL: 2.278472900390625   tA: 0.14217498898506165         vL:
2.277625799179077   vA: 0.1444000005722046
E: 1    tL: 2.0832886695861816  tA: 0.21222499012947083         vL:
2.0869710445404053  vA: 0.21359999477863312
E: 2    tL: 2.0263068675994873  tA: 0.2530499994754791  vL: 2.0331838130950928
vA: 0.25119999051094055
E: 3    tL: 1.968971848487854   tA: 0.2750999927520752  vL: 1.9787909984588623
vA: 0.2685999870300293
E: 4    tL: 1.9420374631881714  tA: 0.29794999957084656         vL:
1.9531933069229126  vA: 0.2920999825000763
E: 5    tL: 1.9121125936508179  tA: 0.3107749819755554  vL: 1.924224615097046
vA: 0.30410000681877136
E: 6    tL: 1.8649016618728638  tA: 0.32850000262260437         vL:
1.8782538175582886  vA: 0.3212999999523163
E: 7    tL: 1.8164889812469482  tA: 0.3481749892234802  vL: 1.8317759037017822
vA: 0.3409000039100647
E: 8    tL: 1.7772924900054932  tA: 0.3669999837875366  vL: 1.7953062057495117
vA: 0.3572999835014343
E: 9    tL: 1.7397589683532715  tA: 0.3808249831199646  vL: 1.7609741687774658
vA: 0.37209999561309814
E: 10   tL: 1.7067975997924805  tA: 0.3930499851703644  vL: 1.7314820289611816
vA: 0.38449999690055847
E: 11   tL: 1.6775034666061401  tA: 0.40414997935295105         vL:
1.7058568000793457  vA: 0.3905999958515167
E: 12   tL: 1.650678277015686   tA: 0.41349998116493225         vL:
1.6829472780227661  vA: 0.40059998631477356
E: 13   tL: 1.6289818286895752  tA: 0.42114999890327454         vL:
1.6646416187286377  vA: 0.407399982213974
E: 14   tL: 1.6080230474472046  tA: 0.4297249913215637  vL: 1.6473249197006226
vA: 0.41449999809265137
E: 15   tL: 1.5864176750183105  tA: 0.4385499954223633  vL: 1.6296629905700684
vA: 0.42319998145103455
```

```
E: 16   tL: 1.5719285011291504  tA: 0.44347497820854187         vL:
1.6187530755996704  vA: 0.42579999566078186
E: 17   tL: 1.5556085109710693  tA: 0.45159998536109924         vL:
1.6060779094696045  vA: 0.4324999749660492
E: 18   tL: 1.5399166345596313  tA: 0.4574749767780304  vL: 1.5948086977005005
vA: 0.44039997458457947
E: 19   tL: 1.5249499082565308  tA: 0.46265000104904175         vL:
1.58340322971344    vA: 0.4438999891281128
E: 20   tL: 1.5090336799621582  tA: 0.4682749807834625  vL: 1.572184681892395
vA: 0.44829997420310974
E: 21   tL: 1.4942880868911743  tA: 0.47269999980926514         vL:
1.5616765022277832  vA: 0.44849997758865356
E: 22   tL: 1.4808235168457031  tA: 0.4781249761581421  vL: 1.5525765419006348
vA: 0.4519999921321869
E: 23   tL: 1.4645767211914062  tA: 0.4840250015258789  vL: 1.5414317846298218
vA: 0.45719999074935913
E: 24   tL: 1.4479894638061523  tA: 0.4894999861717224  vL: 1.5305671691894531
vA: 0.46149998903274536
E: 25   tL: 1.4361202716827393  tA: 0.493149995803833   vL: 1.5239161252975464
vA: 0.4624999761581421
E: 26   tL: 1.4188899993896484  tA: 0.5005499720573425  vL: 1.5127066373825073
vA: 0.4657000005245209
E: 27   tL: 1.404828429222107   tA: 0.5052750110626221  vL: 1.5047402381896973
vA: 0.47119998931884766
E: 28   tL: 1.3929346799850464  tA: 0.5085999965667725  vL: 1.4980820417404175
vA: 0.4721999764442444
E: 29   tL: 1.3696414232254028  tA: 0.5182749629020691  vL: 1.4816571474075317
vA: 0.47859999537467957
E: 30   tL: 1.3558194637298584  tA: 0.5236250162124634  vL: 1.4731965065002441
vA: 0.48069998621940613
E: 31   tL: 1.339540958404541   tA: 0.5292750000953674  vL: 1.4652029275894165
vA: 0.48429998755455017
E: 32   tL: 1.327163815498352   tA: 0.5330749750137329  vL: 1.4585258960723877
vA: 0.487199991941452
E: 33   tL: 1.3164252042770386  tA: 0.5364499688148499  vL: 1.4545962810516357
vA: 0.48799997568130493
E: 34   tL: 1.298164963722229   tA: 0.5413749814033508  vL: 1.4455015659332275
vA: 0.4916999936103821
E: 35   tL: 1.2883391380310059  tA: 0.5440499782562256  vL: 1.4426584243774414
vA: 0.4912000000476837
E: 36   tL: 1.2753905057907104  tA: 0.548799991607666   vL: 1.4392447471618652
vA: 0.4921000003814697
E: 37   tL: 1.2615435123443604  tA: 0.5529749989509583  vL: 1.4343303442001343
vA: 0.4940999746322632
E: 38   tL: 1.2520612478256226  tA: 0.5554749965667725  vL: 1.4325724840164185
vA: 0.49149999022483826
E: 39   tL: 1.2406355142593384  tA: 0.5584749579429626  vL: 1.4311285018920898
vA: 0.4918999969959259
```

```
E: 40   tL: 1.2233253717422485  tA: 0.5666999816894531  vL: 1.4223697185516357
vA: 0.49609997868537903
E: 41   tL: 1.2063863277435303  tA: 0.5703749656677246  vL: 1.4170548915863037
vA: 0.49559998512268066
E: 42   tL: 1.197527527809143   tA: 0.5738499760627747  vL: 1.4165488481521606
vA: 0.4982999861240387
E: 43   tL: 1.190682053565979   tA: 0.5746999979019165  vL: 1.420558214187622
vA: 0.49449998140335083
E: 44   tL: 1.1825757026672363  tA: 0.5766249895095825  vL: 1.420655608177185
vA: 0.49469998478889465
E: 45   tL: 1.1709179878234863  tA: 0.5788750052452087  vL: 1.4202147722244263
vA: 0.49459999799728394
E: 46   tL: 1.1652510166168213  tA: 0.5796499848365784  vL: 1.4253321886062622
vA: 0.4948999881744385
E: 47   tL: 1.1514488458633423  tA: 0.5859999656677246  vL: 1.4215196371078491
vA: 0.49539998173713684
E: 48   tL: 1.148616075515747   tA: 0.5854249596595764  vL: 1.4280489683151245
vA: 0.49629998207092285
E: 49   tL: 1.1397567987442017  tA: 0.5858500003814697  vL: 1.4298659563064575
vA: 0.4948999881744385
CPU times: user 36min 29s, sys: 1min 7s, total: 37min 37s
Wall time: 37min 37s
```

5.1 Statistics

Final Training Loss: 1.140

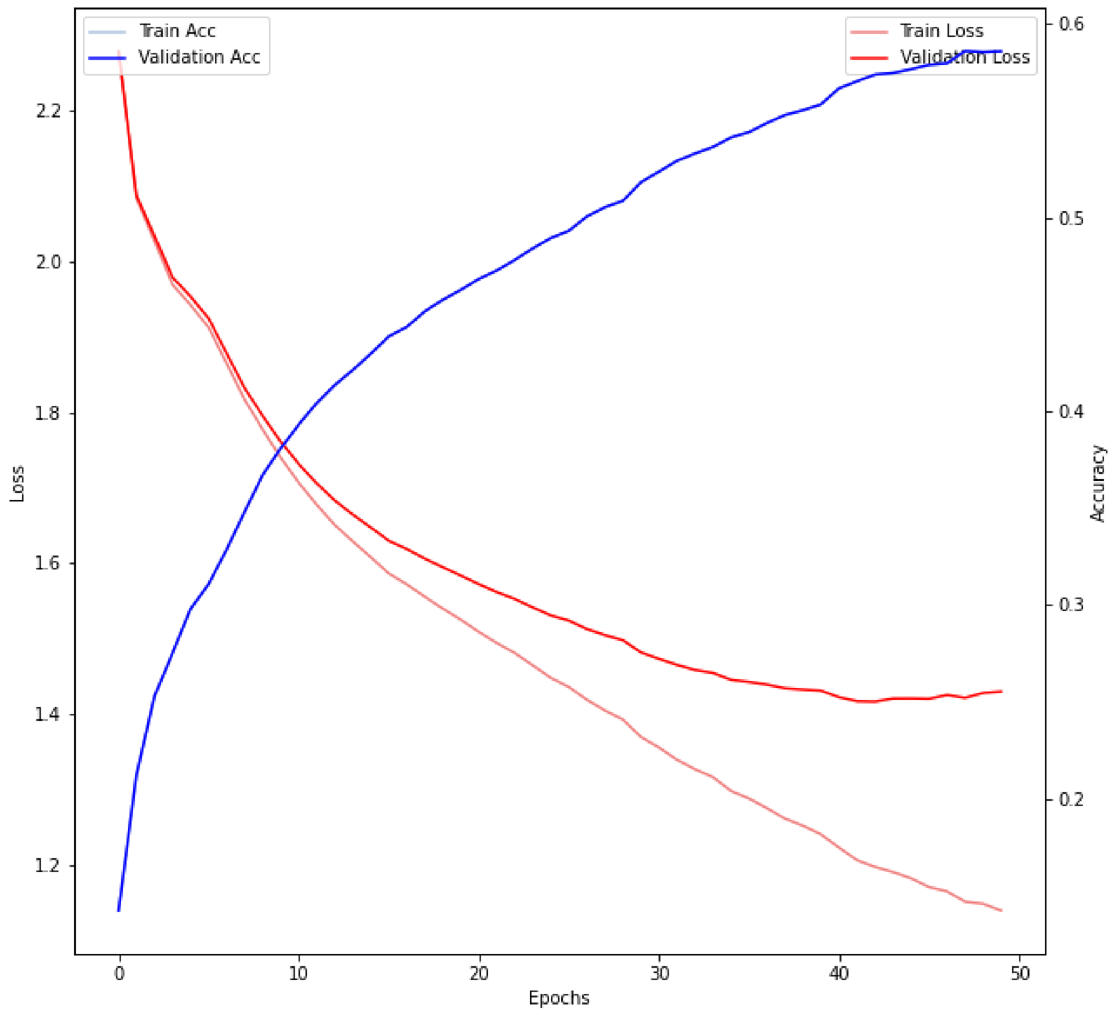
Final Training Accuracy: 0.586

Final Validation Loss: 1.43

Final Validation Accuracy: 0.495

5.1.1 Training Graph

[9]:
```
fig = plt.figure(figsize=(10,10))
pls = np.array(train_data)
ax = sns.lineplot(x=pls[:,0], y=pls[:,1], color="lightcoral", label="Train␣
 ↪Loss")
ax = sns.lineplot(x=pls[:,0], y=pls[:,3], color="r", label="Validation Loss")
ax.set_xlabel("Epochs")
ax.set_ylabel("Loss")
ax2 = plt.twinx()
ax2.set_ylabel("Accuracy")
sns.lineplot(x=pls[:,0], y=pls[:,2], color="lightsteelblue", label="Train Acc")
sns.lineplot(x=pls[:,0], y=pls[:,2], color="b", label="Validation Acc")
plt.show()
```



6 Reflection / Discussion

6.1 Start of the network

The initial lab felt very odd in this series. The client, how it was suggested to be implemented, still doesn't quite make sense to me. Hence I have likely a different client implementation than most other people.

6.1.1 Design Philosophy

The suggestion to manually create the layers and link them together I think in some ways was a fine one. My lab 3 I believe had basically this feature. But very quickly I struggled with modifying the layers, linking them to the output, and deciding what to do with them. So what I ended up doing was simply making a driver, in the form of the NeuralNetwork class, to handle that for me.

My gradient tape was a linked list with manual calls to the regulaization layers. This made more sense to me then doing an array, which I did in lab 3 and was one of the reasons why I thought that lab to be very tedious.

Because of this, I certainly had obfuscated information. It would be neigh impossible to call a layer mid-inference, and difficult to show the structure of the network without just showing the code that made it.

But because the wrappers were so comprehensive in my network, it meant I can feasibly train on any dataset with any amount of layers, types of layers, and size of layers without much issue. Adding types layers is just adding the layer in layers then adding a wrapper, easy as that.

My client was basically what you'd get with a scikit learn wrapper. Train, train test split, inference, and statistics. Not much more, not much less.

6.2 Back Prop

I thought the experience deriving the back propogation was fine. I do wish there was a little more guidance, and, moreso than guidance, specific suggestions on my original backprop that would say 'this spot is wrong or will be hard to translate due to unclear stuff.' I saw that first hand when I began unit testing, especially against pytorch's autograd, where and how some of my gradients were just minimally wrong.

More time needed to be put into a single layer backprop instead of going through the whole network in week 3-5 and calculating each layer in a sludge of information. My fist quiz on backprop I failed (9/20?) yet the second, which was a test, I think I nearly aced. It was connecting the dots from the sludge of information where providing the dots would have been more effective.

6.3 Unit Tests

The unit tests have me split. Yes, it is good to do them. But at the same time, my unit tests I am turning in now *do not work*. It is not because my equations are wrong, but it's because I had to change a couple things from the toy unit tests to an actual training scenerio in my Network class that broke the unit tests. Could I have seen that coming, yes, but that was the case for how my implementation went.

6.4 Final Curves

My network seems to be performing as I would expect a normal large neural network to perform. There is overfitting, yes, but the loss, both validation and training, are going down at a lessening rate each epoch. Later epochs had some odd overfitting on both datasets (CIFAR/MNIST).

6.5 Final Notes

This series was an extremely fun, but also sometimes frustating series. The abstraction of information was made easier to debug with the unit tests, but the original design was tedious and annoying to debug as well. The backprop is not necessarily stable, but with the right starting values, results in the current training that seems to perform well. Overall, one of my favorite lab series I have run.