Question 1

In [200…

```
import pickle
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from sklearn.model_selection import KFold
from torch.utils.data import Dataset, DataLoader
from tqdm import tqdm
import time
```

In [201…

```
def timeit(f):

    def timed(*args, **kw):

        ts = time.time()
        result = f(*args, **kw)
        te = time.time()

        print(f'func:{f.__name__} took: {te-ts:.4f} sec')
        return result

    return timed
```

In [2]:

```
def load_dataset(path):
    with open(path, 'rb') as f:
        train_data, test_data = pickle.load(f)

    X_train = torch.tensor(train_data[0], dtype=torch.float)
    y_train = torch.tensor(train_data[1], dtype=torch.long)
    X_test = torch.tensor(test_data[0], dtype=torch.float)
    y_test = torch.tensor(test_data[1], dtype=torch.long)
    return X_train, y_train, X_test, y_test
```

In [3]:

```
class MnistDataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y

    def __len__(self):
        return len(self.y)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

(a)

In [4]:

```
X_train, y_train, X_test, y_test = load_dataset("mnist.pkl")
print("X_train shape:", X_train.shape)
```

```
print("X_test shape:", X_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape:", y_test.shape)
```

```
X_train shape: torch.Size([60000, 32, 32])
X_test shape: torch.Size([10000, 32, 32])
y_train shape: torch.Size([60000])
y_test shape: torch.Size([10000])
```

In [158…

```
X_train_norm = X_train.reshape(X_train.shape[0], -1) / torch.max(X_train)
X_test_norm = X_test.reshape(X_test.shape[0], -1) / torch.max(X_test)
# X_train_norm = X_train.flatten(d
print(X_train_norm.shape)
```

```
torch.Size([60000, 1024])
```

In [41]:

```
# # Normalization to the maximum pixel value of the dataset

# X_train_norm = X_train / torch.max(X_train)
# X_test_norm = X_test / torch.max(X_test)
# print(X_train_norm.shape)
# print(X_train_norm.shape)
```

```
torch.Size([60000, 32, 32])
torch.Size([60000, 32, 32])
```

Below is me testing how to divide each image by their respective maximum pixel value, even though we know for this case that the max pixel value will be 255. In the scenario that the maximums would differ, one would like to sequentially divide each image by its own maximum pixel value.

In [5]:

```
# # First find the maxes of each row of each of the 2D tensors
# values, indices = torch.max(X_train, dim=2)
# display(values)
# display(values.shape)
# display(values[0])
```

```
tensor([[0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.]])
torch.Size([60000, 32])
tensor([  0.,   0.,   0.,   0.,   0.,   0.,   0., 255., 253., 253., 253., 25
3.,
        253., 253., 253., 241., 253., 253., 253., 253., 253., 253., 253., 25
3.,
        253., 253., 253.,   0.,   0.,   0.,   0.,   0.])
```

In [11]:

```
# # Next, find the single max value of each 2D tensor
# maxes, indices_2 = torch.max(values, dim=1)
# display(maxes.shape)
# maxes
```

```
torch.Size([60000])
```

Out[11]:
```
tensor([255., 255., 255.,  ..., 255., 255., 255.])
```

In [35]:
```
# maxes1D = maxes.unsqueeze(1)
# display(maxes1D.shape)
# display(maxes1D)
```

```
torch.Size([60000, 1])
tensor([[255.],
        [255.],
        [255.],
        ...,
        [255.],
        [255.],
        [255.]])
```

In [7]:
```
# maxes1D = maxes.unsqueeze(1).repeat_interleave(repeats=32, dim=1)
# display(maxes1D.shape)
# display(maxes1D)
```

```
torch.Size([60000, 32])
tensor([[255., 255., 255.,  ..., 255., 255., 255.],
        [255., 255., 255.,  ..., 255., 255., 255.],
        [255., 255., 255.,  ..., 255., 255., 255.],
        ...,
        [255., 255., 255.,  ..., 255., 255., 255.],
        [255., 255., 255.,  ..., 255., 255., 255.],
        [255., 255., 255.,  ..., 255., 255., 255.]])
```

In [8]:
```
# maxes2D = maxes1D.repeat(1, 60000)
```

In [9]:
```
# test = maxes1D[0].repeat(32, 1)
# display(test.shape)
# test
```

```
torch.Size([32, 32])
```

Out[9]:
```
tensor([[255., 255., 255.,  ..., 255., 255., 255.],
        [255., 255., 255.,  ..., 255., 255., 255.],
        [255., 255., 255.,  ..., 255., 255., 255.],
        ...,
        [255., 255., 255.,  ..., 255., 255., 255.],
        [255., 255., 255.,  ..., 255., 255., 255.],
        [255., 255., 255.,  ..., 255., 255., 255.]])
```

In [ ]:
```
# maxes2D = maxes1D.repeat(60000, 32, 1)
# display(maxes2D.shape)
```

In [39]:
```
# X_train_norm = X_train / maxes2D
```

(b)

Complete the following Python class for training/evaluation

In [169…
```
import numpy as np
from tqdm import tqdm
```

```
class Trainer:

    def __init__(self, model, opt_method, learning_rate, batch_size, epoch,
        self.model = model

        if opt_method == "adam":
            self.optimizer = torch.optim.Adam(model.parameters(), learning_r
        else:
            raise NotImplementedError("This optimization is not supported")

        self.epoch = epoch
        self.batch_size = batch_size

    def train(self, train_data, val_data, early_stop=True, verbose=True, dra
        train_loader = DataLoader(train_data, batch_size=self.batch_size, sh

        train_loss_list, train_acc_list = [], []
        val_loss_list, val_acc_list = [], []
        weights = self.model.state_dict()
        lowest_val_loss = np.inf
        loss_func = nn.CrossEntropyLoss()
        for n in tqdm(range(self.epoch), leave=False):
            # enable train mode
            self.model.train()
            epoch_loss, epoch_acc = 0.0, 0.0
            for X_batch, y_batch in train_loader:
                # batch_importance is the ratio of batch_size
                batch_importance = y_batch.shape[0]/len(train_data)
                y_pred = self.model(X_batch)
                batch_loss = loss_func(y_pred, y_batch)

                self.optimizer.zero_grad()
                batch_loss.backward()
                self.optimizer.step()

                epoch_loss += batch_loss.detach().cpu().item() * batch_impor
                batch_acc = torch.sum(torch.argmax(y_pred, axis=1) == y_batc
                epoch_acc += batch_acc.detach().cpu().item() * batch_importa
            train_loss_list.append(epoch_loss)
            train_acc_list.append(epoch_acc)
            val_loss, val_acc = self.evaluate(val_data)
            val_loss_list.append(val_loss)
            val_acc_list.append(val_acc)

            if early_stop:
                if val_loss < lowest_val_loss:
                    lowest_val_loss = val_loss
                    weights = self.model.state_dict()

        if draw_curve:
            x_axis = np.arange(self.epoch)
            fig, axes = plt.subplots(1, 2, figsize=(10, 4))
            axes[0].plot(x_axis, train_loss_list, label="Train")
            axes[0].plot(x_axis, val_loss_list, label="Validation")
            axes[0].set_title("Loss")
```

```
            axes[0].legend()
            axes[1].plot(x_axis, train_acc_list, label='Train')
            axes[1].plot(x_axis, val_acc_list, label='Validation')
            axes[1].set_title("Accuracy")
            axes[1].legend()

        if early_stop:
            self.model.load_state_dict(weights)

        return {
            "train_loss_list": train_loss_list,
            "train_acc_list": train_acc_list,
            "val_loss_list": val_loss_list,
            "val_acc_list": val_acc_list,
        }

    def evaluate(self, data, print_acc=False):
        # enable evaluation mode
        self.model.eval()
        loader = DataLoader(data, batch_size=self.batch_size, shuffle=True)
        loss_func = nn.CrossEntropyLoss()
        acc, loss = 0.0, 0.0
        for X_batch, y_batch in loader:
            with torch.no_grad():
                batch_importance = y_batch.shape[0]/len(data)
                y_pred = self.model(X_batch)
                batch_loss = loss_func(y_pred, y_batch)
                batch_acc = torch.sum(torch.argmax(y_pred, axis=1) == y_batc
                acc += batch_acc.detach().cpu().item() * batch_importance
                loss += batch_loss.detach().cpu().item() * batch_importance
        if print_acc:
            print(f"Accuracy: {acc:.3f}")
        return loss, acc
```

Complete the following function to do KFold cross validation

In [198…

```
@timeit
def KFoldCrossValidation(
    model_class, k,
    X_train, y_train, X_test, y_test,
    opt_method='adam', learning_rate=2e-3, batch_size=128, epoch=50, l2=0.0
):
    # Use MnistDataset to organize data
    test_data = MnistDataset(X_test, y_test)
    kf = KFold(n_splits=k, shuffle=True, random_state=12)
    train_acc_list, test_acc_list = [], []
    for i, (train_index, val_index) in enumerate(kf.split(X_train)):
        print(f"Fold {i}:")

        # Use MnistDataset to organize data

        train_data = MnistDataset(X_train[train_index], y_train[train_index]
        val_data = MnistDataset(X_train[val_index], y_train[val_index])

        model = model_class()
```

```
        # initialize a Trainer object
        trainer = Trainer(model, 'adam', learning_rate, batch_size, epoch, l
        # call trainer.train() here
        res = trainer.train(train_data, val_data)
        # record the training accuracy of the epoch that has the lowest vali
        # Hint: use np.argmin
        train_acc_best = res['train_acc_list'][np.argmin(res['val_loss_list'
        # test, use trainer.evaluate function
        test_loss, test_acc = trainer.evaluate(test_data)

        train_acc_list.append(train_acc_best)
        test_acc_list.append(test_acc)

        print(f"Training accuracy: {train_acc_best}")
        print(f"Test accuracy: {test_acc}")

    print("Final results:")
    # Report mean and std
    print(f"Training accuracy:, {np.mean(train_acc_list)}, std: {np.std(trai
    print(f"Test accuracy:, {np.mean(test_acc_list)}, std: {np.std(test_acc_
```

(c)

In [171…

```
class Net3(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(1024, 3),
            nn.Sigmoid(),
            nn.Linear(3, 10),
            nn.Sigmoid(),
        )

    def forward(self, x):
        return self.layers(x)
```

In [172…

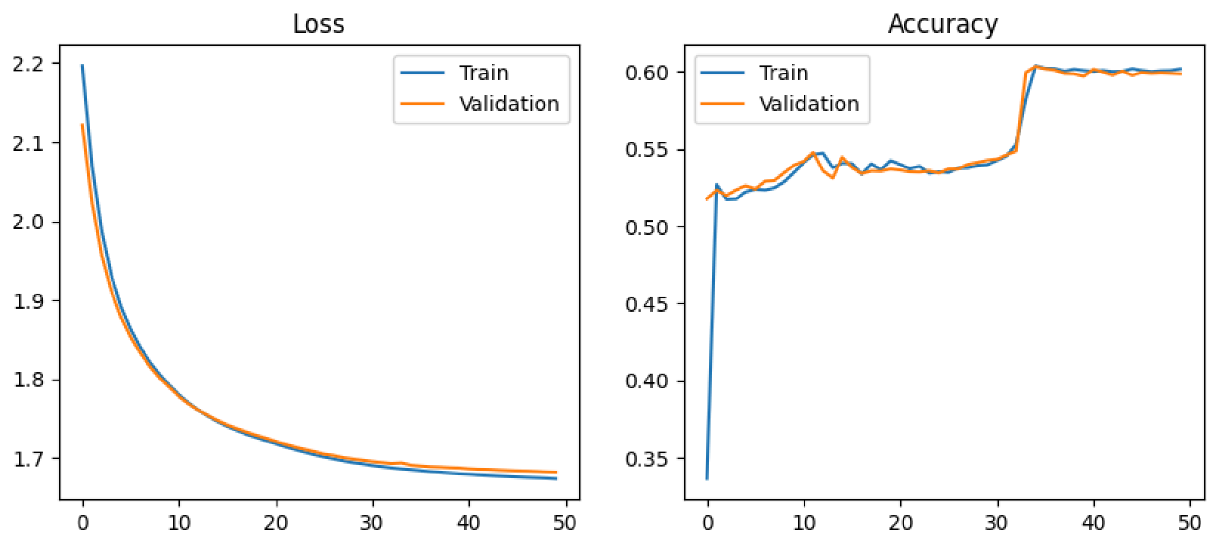
```
train_data = MnistDataset(X_train, y_train)
test_data = MnistDataset(X_test, y_test)
train_loader = DataLoader(train_data, batch_size=128, shuffle=True)
test_loader = DataLoader(test_data, batch_size=128, shuffle=True)
```

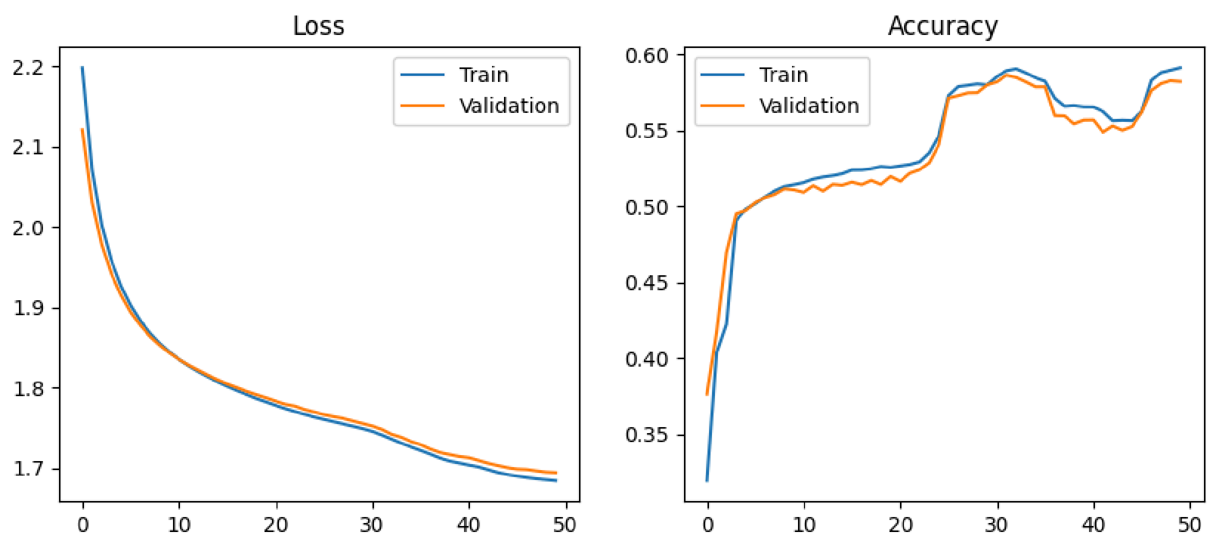
In [173…

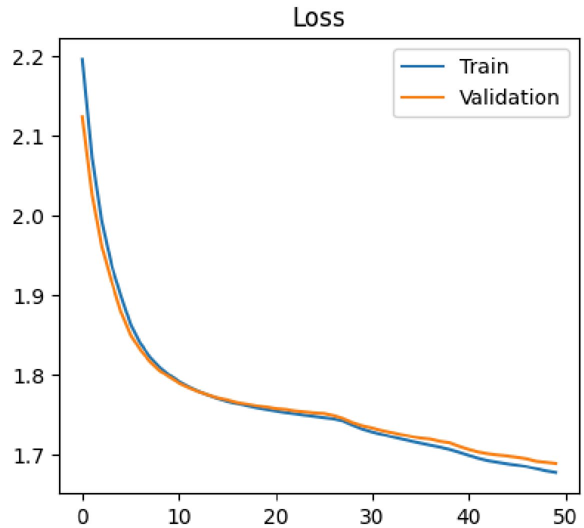
```
model = Net3()
KFoldCrossValidation(Net3,
                     k=3,
                     X_train=X_train_norm,
                     y_train=y_train,
                     X_test=X_test_norm,
                     y_test=y_test,
                     opt_method='adam',
                     learning_rate=2e-3,
                     batch_size=128,
                     epoch=50,
```

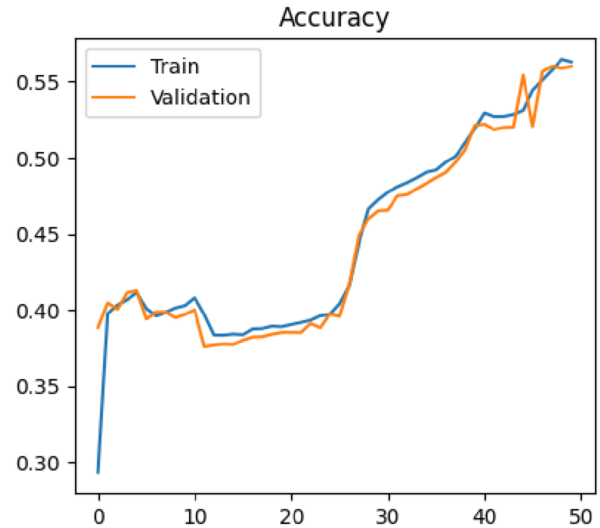
```
                    l2=0.0
                   )
```

```
Fold 0:

Training accuracy: 0.601725000000001
Test accuracy: 0.6002999999999998
Fold 1:

Training accuracy: 0.5910500000000002
Test accuracy: 0.5819
Fold 2:

Training accuracy: 0.5629500000000004
Test accuracy: 0.5734999999999999
Final results:
Training accuracy:, 0.5852416666666671, std: 0.016353953820271137
Test accuracy:, 0.5852333333333333, std: 0.011192060678097773
```









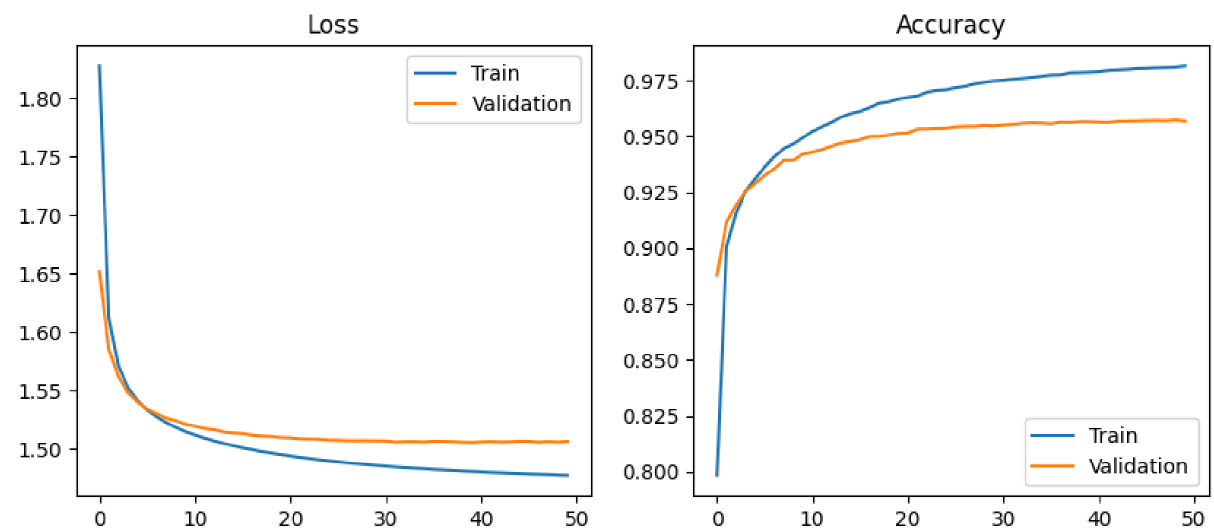
With a neural network architecture of only having a hidden layer consisting of 3 neurons, the bias is quite high and the variance is quite low. This is because there are not as many neurons in the hidden layer to add increased dimensionality to our model, and thus the accuracy is quite low. Thus, for the 3-neuron hidden layer, the bias is higher and the variance is lower (the accuracy and loss look quite similar for both the training and validation sets).

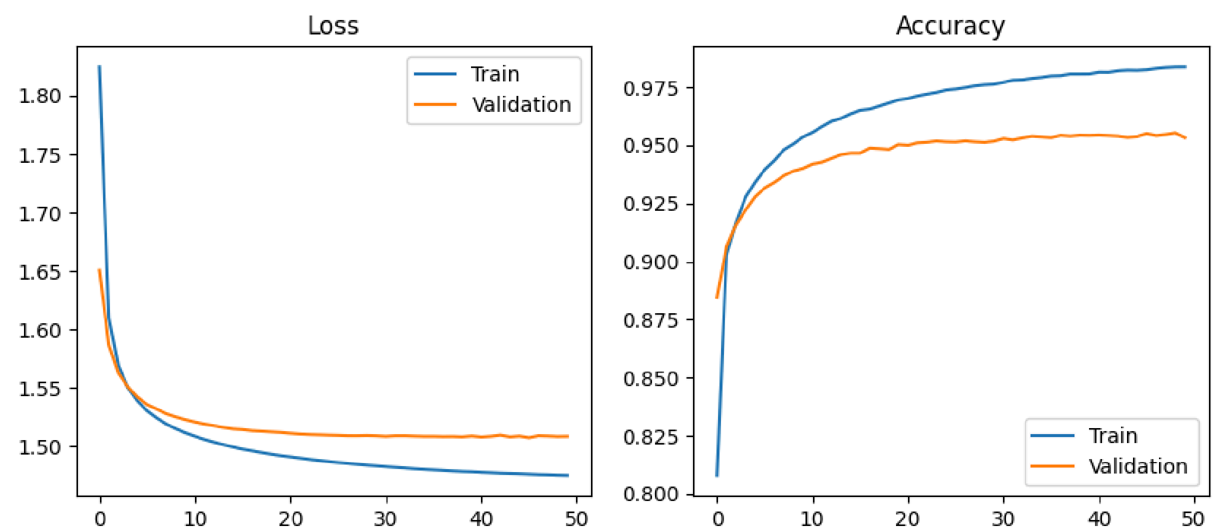
(d)

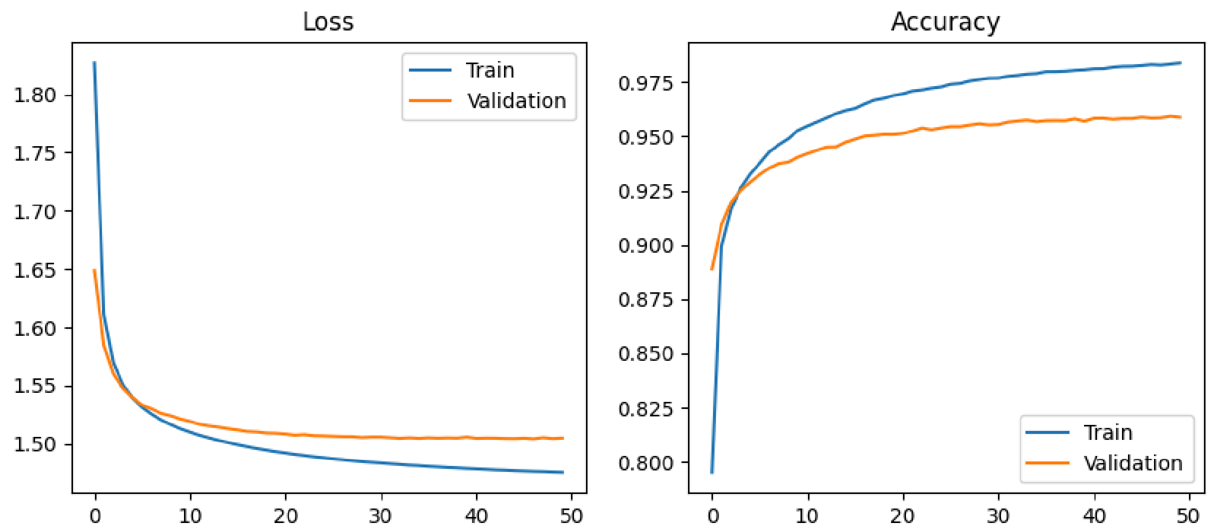
In [174…
```
class Net50(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(1024, 50),
            nn.Sigmoid(),
            nn.Linear(50, 10),
            nn.Sigmoid(),
        )

    def forward(self, x):
        return self.layers(x)
```

In [176…
```
model = Net50()
KFoldCrossValidation(Net50,
                     k=3,
                     X_train=X_train_norm,
                     y_train=y_train,
                     X_test=X_test_norm,
                     y_test=y_test,
                     opt_method='adam',
                     learning_rate=2e-3,
                     batch_size=128,
                     epoch=50,
                     l2=0.0
                    )
```

```
Fold 0:

Training accuracy: 0.9787250000000012
Test accuracy: 0.9575999999999997
Fold 1:

Training accuracy: 0.9824500000000009
Test accuracy: 0.9588999999999993
Fold 2:

Training accuracy: 0.9830500000000013
Test accuracy: 0.9575999999999995
Final results:
Training accuracy:, 0.9814083333333344, std: 0.0019131489458191347
Test accuracy:, 0.9580333333333328, std: 0.0006128258770282213
```







For a hidden layer with 50 neurons, the bias is significantly less but the variance is also much higher. The accuracy increased, but the difference in loss/accuracy between the training and validation sets is much larger than 3-neuron hidden layer.

Question 2

(a)

In [178…

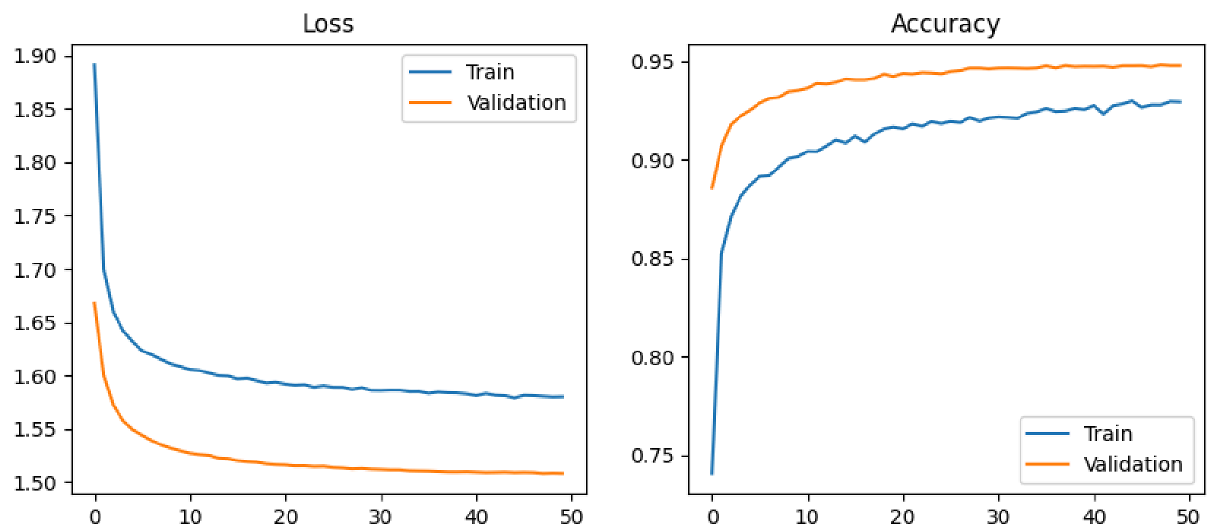
```
class Net50Dropout(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(1024, 50),
            nn.Dropout(p=0.15),
            nn.Sigmoid(),
            nn.Linear(50, 10),
            nn.Dropout(p=0.1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.layers(x)
```

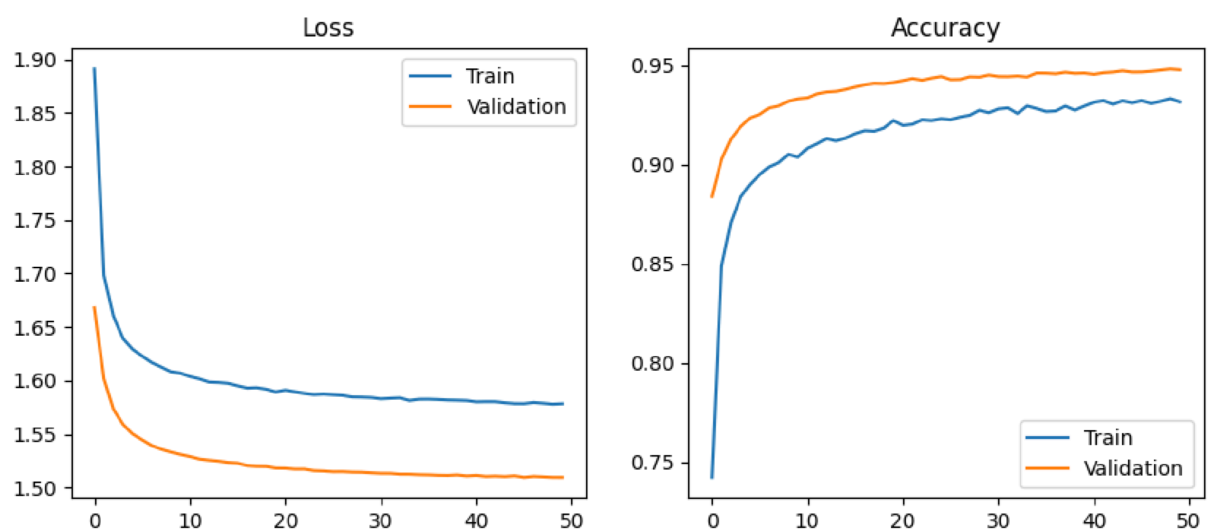
In [179…

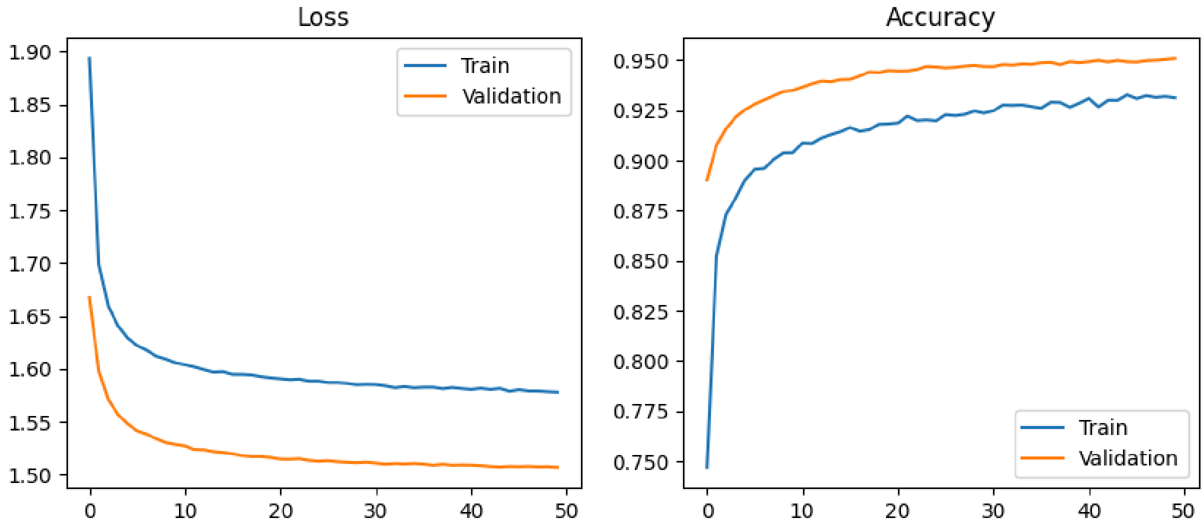
```
KFoldCrossValidation(Net50Dropout,
                     k=3,
                     X_train=X_train_norm,
                     y_train=y_train,
                     X_test=X_test_norm,
                     y_test=y_test,
                     opt_method='adam',
                     learning_rate=2e-3,
                     batch_size=128,
                     epoch=50,
```

```
                    l2=0.0
                   )
```

```
Fold 0:

Training accuracy: 0.9279250000000001
Test accuracy: 0.9488999999999999
Fold 1:

Training accuracy: 0.9316500000000012
Test accuracy: 0.9496999999999998
Fold 2:

Training accuracy: 0.9312750000000003
Test accuracy: 0.9505999999999996
Final results:
Training accuracy:, 0.9302833333333339, std: 0.0016746060896690512
Test accuracy:, 0.9497333333333331, std: 0.0006944222218665326
```







Interestingly, compared to 1d), the test accuracy is greater than the training accuracy, and vice versa for the loss: the validation loss is less than the training loss. In 1d), the training accuracy was always higher than the validation accuracy, and the training loss ended up lower than the validation loss. Here, it is flipped. However, the values for test accuracy using dropout are still a bit lower than the values for test accuracy for Net50 without dropout. Additionally, because the training accuracy is lower than the test accuracy, the training accuracy with dropout is also lower than without dropout.

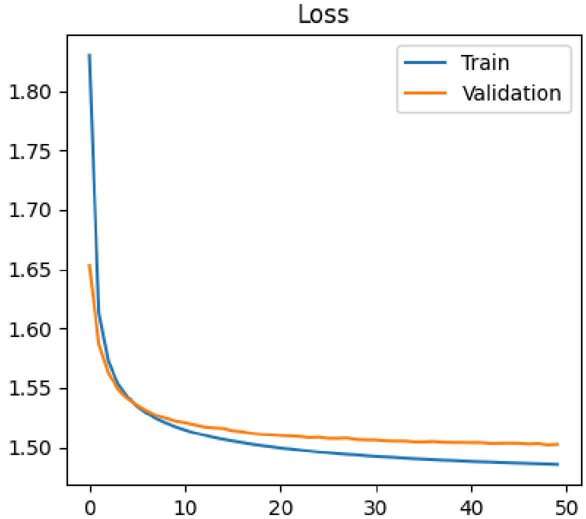
(b)

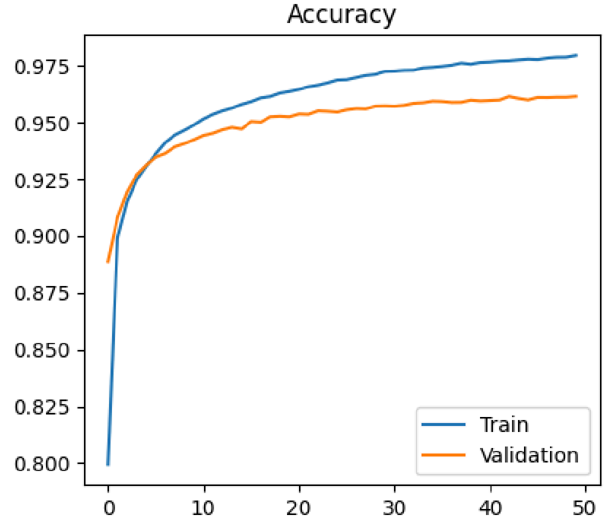
In [180…

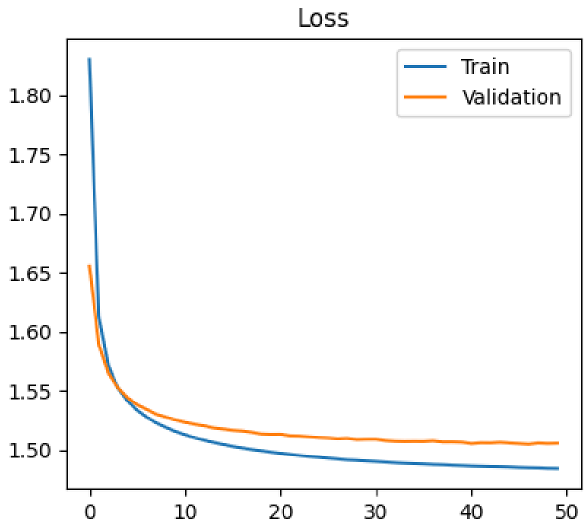
```
# L2 Regularizaiton by setting the "l2" parameter in KFoldCrossValidation
KFoldCrossValidation(Net50,
                     k=3,
                     X_train=X_train_norm,
                     y_train=y_train,
                     X_test=X_test_norm,
                     y_test=y_test,
                     opt_method='adam',
                     learning_rate=2e-3,
                     batch_size=128,
                     epoch=50,
                     l2=1e-5
)
```

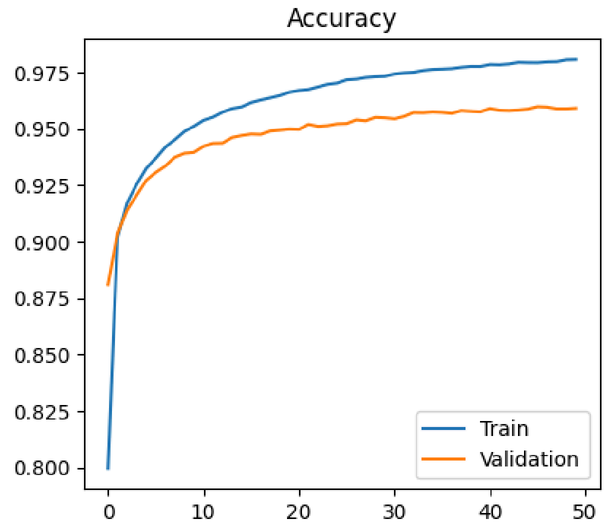
```
Fold 0:

Training accuracy: 0.9786500000000014
Test accuracy: 0.9662999999999999
Fold 1:

Training accuracy: 0.9795750000000006
Test accuracy: 0.9632999999999993
Fold 2:
```

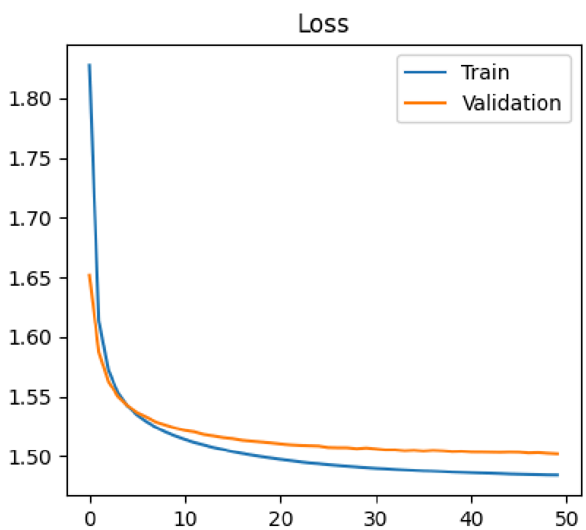
```
Training accuracy: 0.9808500000000007
Test accuracy: 0.9631999999999992
Final results:
Training accuracy:, 0.9796916666666675, std: 0.0009019269494929994
Test accuracy:, 0.9642666666666658, std: 0.0014383632673593493
```

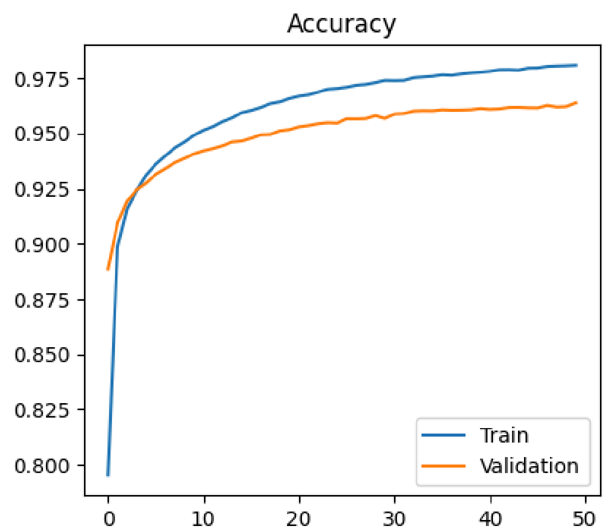












Using L2 regularization, the both the training and validation accuracy increased compared to the Net50 with Dropout model. The training accuracy is still lower than the training accuracy of the original Net50 model, but the validation accuracy improved compared to the original Net50 model (from 95.8% to 96.4%)

(c)

For debugging: You should get 331 features.

In [187…
```
X_train, y_train, X_test, y_test = load_dataset("mnist.pkl")
print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape:", y_test.shape)
```

```
X_train shape: torch.Size([60000, 32, 32])
X_test shape: torch.Size([10000, 32, 32])
y_train shape: torch.Size([60000])
y_test shape: torch.Size([10000])
```

In [188…
```
from sklearn.decomposition import PCA

# Flatten the inputs & normalization
X_train = X_train.reshape(X_train.shape[0], -1) / torch.max(X_train)
X_test = X_test.reshape(X_test.shape[0], -1) / torch.max(X_test)
print(X_train.shape)


# keeping specific number of features
pca = PCA(n_components=0.99)
# fit
pca.fit(X_train)
# transform
X_train_pca = torch.tensor(pca.transform(X_train), dtype=torch.float)
X_test_pca = torch.tensor(pca.transform(X_test), dtype=torch.float)
print(X_train_pca.shape, X_test_pca.shape)
```

```
torch.Size([60000, 1024])
torch.Size([60000, 331]) torch.Size([10000, 331])
```

We only have 331 parameters (features) this time, instead of 1024 from the original ANN. Since there are significantly less features, the NN should train faster!

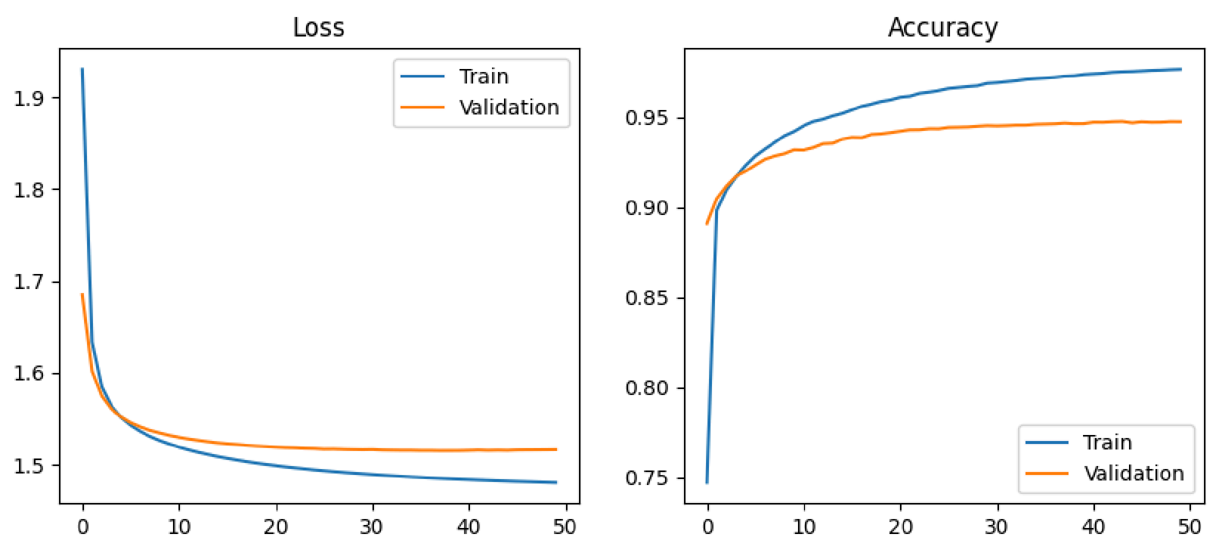
In [191…
```
# Use one hidden layer of size 50, no Dropouts
class Net50PCA(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(331, 50),
            nn.Sigmoid(),
            nn.Linear(50, 10),
            nn.Sigmoid(),
```

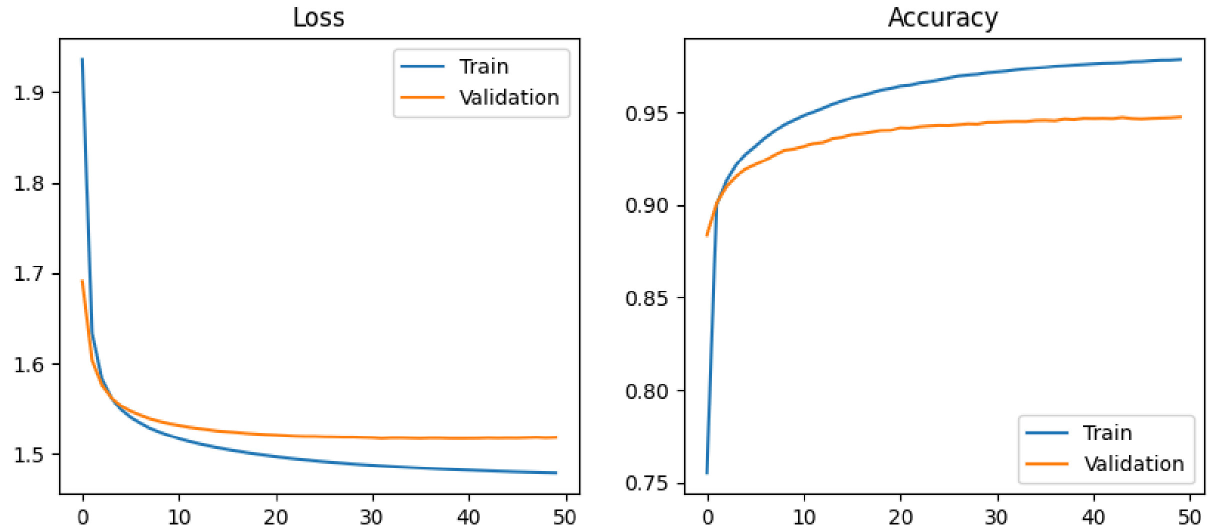
```
        )


    def forward(self, x):
        return self.layers(x)
```

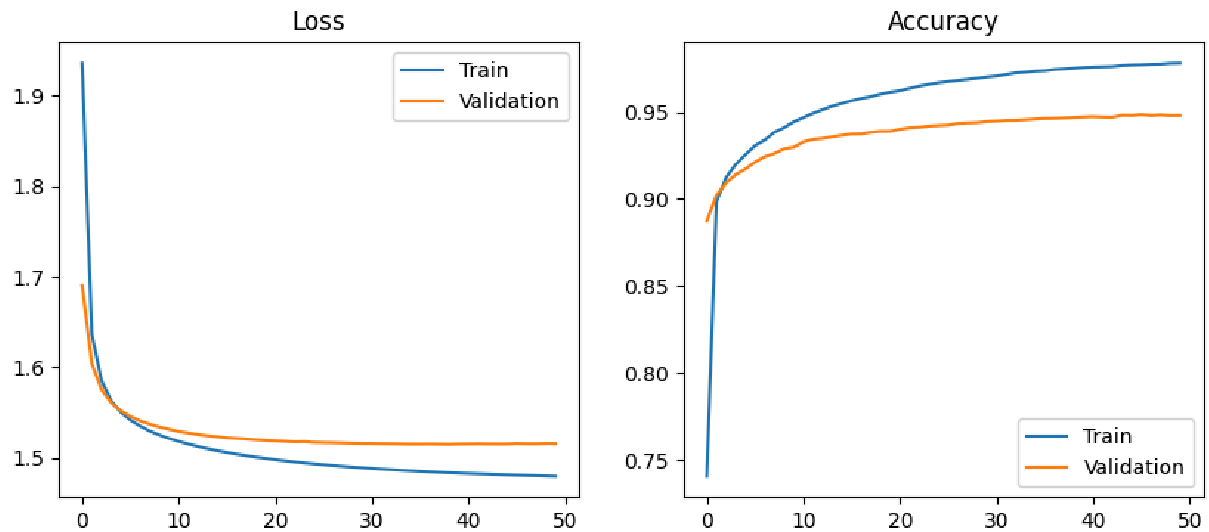
In [203…

```
KFoldCrossValidation(Net50PCA,
                     k=3,
                     X_train=X_train_pca,
                     y_train=y_train,
                     X_test=X_test_pca,
                     y_test=y_test,
                     opt_method='adam',
                     learning_rate=2e-3,
                     batch_size=128,
                     epoch=50,
                     l2=0.0
                    )
```

```
Fold 0:

Training accuracy: 0.9727500000000013
Test accuracy: 0.9491999999999996
Fold 1:

Training accuracy: 0.9759250000000014
Test accuracy: 0.9507999999999995
Fold 2:

Training accuracy: 0.9752250000000015
Test accuracy: 0.9514999999999997
Final results:
Training accuracy:, 0.9746333333333347, std: 0.0013620348339484448
Test accuracy:, 0.9504999999999996, std: 0.0009626352718795976
func:KFoldCrossValidation took: 71.7641 sec
```







The training and test accuracy for the PCA data is very similar to the original, but the accuracies are slightly lower for both training and test. The trends remain the same as the non-PCA data.

(d)

In [ ]:

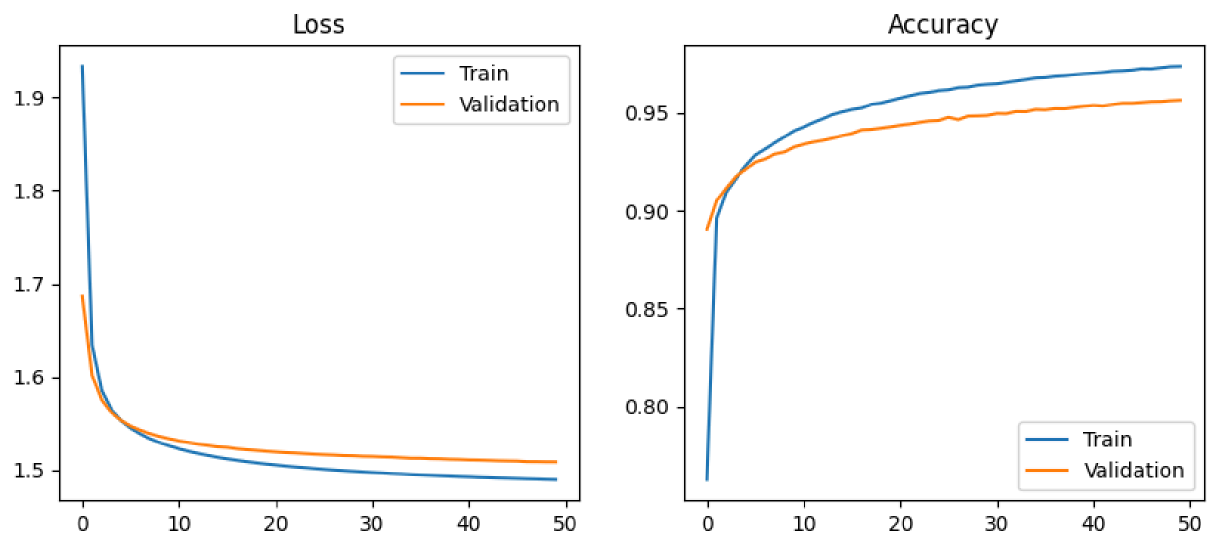
```
# If you find Dropout is better, finish this Net50PCADropout and do K-Fold C

# class Net50PCADropout(nn.module):
#     def __init__(self):
#         super().__init__()
#         ...

#     def forward(self, x):
#         return ...
```

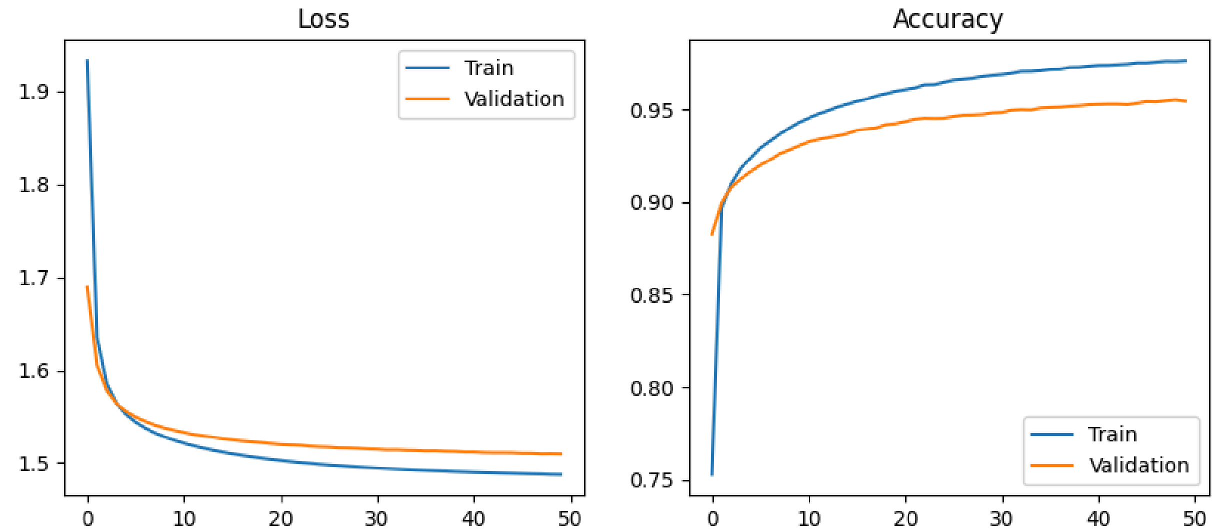
In [202…

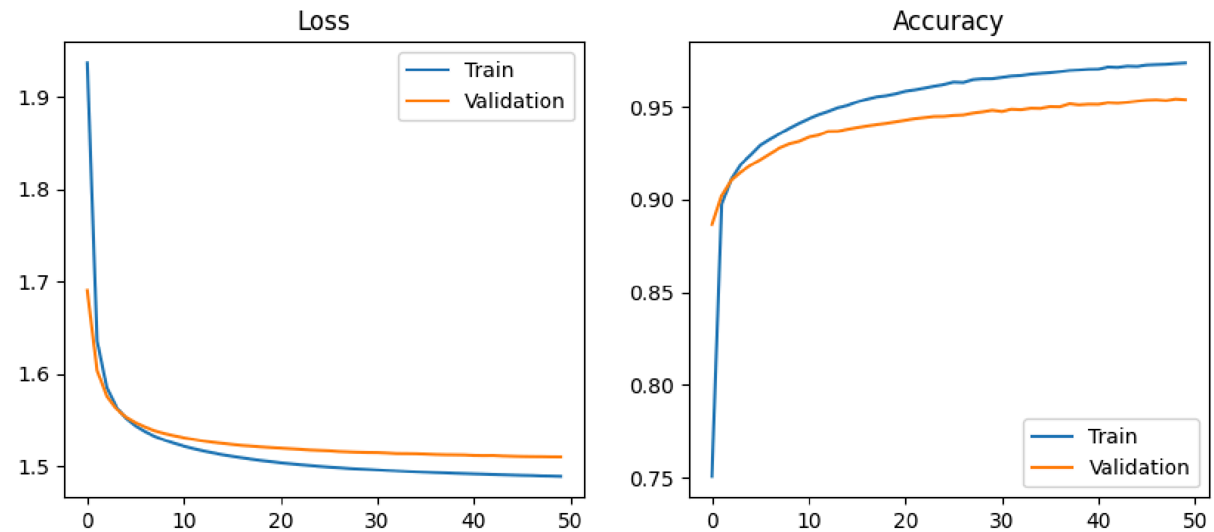
```
# If you find L2 Regularization is better,
# just call KFoldCrossValidation with Net50PCA and l2 set to non-zeros

# L2 Regularizaiton by setting the "l2" parameter in KFoldCrossValidation
```

```
KFoldCrossValidation(Net50PCA,
                     k=3,
                     X_train=X_train_pca,
                     y_train=y_train,
                     X_test=X_test_pca,
                     y_test=y_test,
                     opt_method='adam',
                     learning_rate=2e-3,
                     batch_size=128,
                     epoch=50,
                     l2=1e-5
                    )
```

```
Fold 0:

Training accuracy: 0.973725000000002
Test accuracy: 0.9554999999999996
Fold 1:

Training accuracy: 0.9763750000000015
Test accuracy: 0.9566999999999998
Fold 2:

Training accuracy: 0.9738500000000015
Test accuracy: 0.9561999999999995
Final results:
Training accuracy:, 0.9746500000000017, std: 0.001220826222959852
Test accuracy:, 0.956133333333333, std: 0.0004921607686745203
func:KFoldCrossValidation took: 75.2264 sec
```







The model does train better compared to without L2 regularization. The accuracy for the training data set increased from 0.9746333333333347 to .9746500000000017. The accuracy for the test data set increased from 0.9504999999999996 to 0.9561333333333333. The time taken to train the model increased from 71.7641 sec to 75.2264 sec