Deep Learning (Fall 2023) - Homework 4

Developed by Hongtau Wu & Suzanna Sia. Modified by Ping-Cheng Ku

This notebook contains all starter code for Homework 4. Please read the written assignment carefully to ensure you include all necessary outputs in your final report. Your submission to Homework 4-notebook should include this notebook (.ipynb file), and a PDF (.pdf) of this notebook, and the hw4_utils.py file.

Problem 1a)

Imports

```
## External Libararies
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon
from matplotlib.collections import PatchCollection
```

Data Loading

```
from google.colab import drive
drive.mount('/content/drive/')

# Spectify Path to Provided Data Here
DATA_PATH = '/content/drive/My Drive/fall23_hw4_prob2_data.npy'

## Load Data and Check Dimensionality
data = np.load(DATA_PATH)
Y = data[:,2]
X = data[:,0:2]
print("Y:", Y.shape)
print("X:", X.shape)

## Polygon Boundaries
p = [[[500, 1000], [300, 800], [400, 600], [600, 600], [700, 800]],
     [[500, 600], [100, 400], [300, 200], [700, 200], [900, 400]]]
p = np.asarray(p)
p0 = p[0]
p1 = p[1]
```

```
Mounted at /content/drive/
Y: (60000,)
X: (60000, 2)
```

Visualization Code

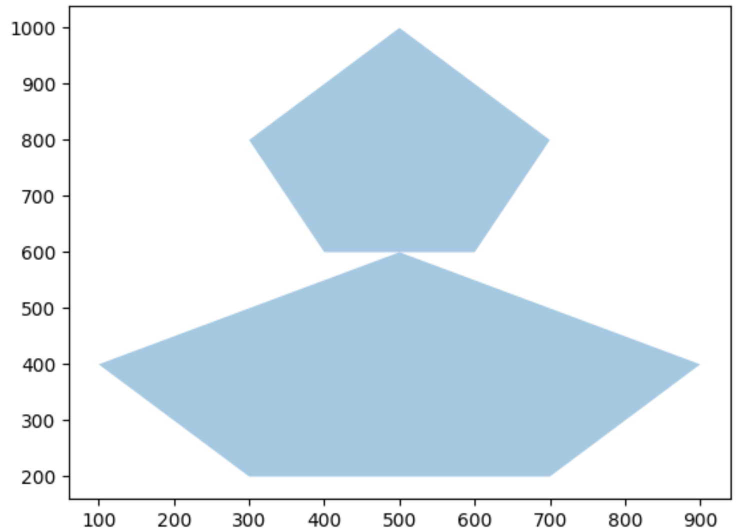
Do not touch any of the visualization code below.

```
## Helper code for visualisation (No Need to Touch)
def visualize_polygons(p0, p1):

    fig, ax = plt.subplots()
    patches = []
    polygon1 = Polygon(p0, True)
    polygon2 = Polygon(p1, True)
    patches.append(polygon1)
    patches.append(polygon2)
    p = PatchCollection(patches, cmap=matplotlib.cm.jet, alpha=0.4)
    ax.add_collection(p)
    ax.autoscale_view()
    plt.show()

def visualize_datapoints(X, Y):

    assert(X.shape[0] == Y.shape[0])
```

```
    fig, ax = plt.subplots()
    npts = 60000
    col = np.where(Y[:npts]==1,'m','b')
    x1 = X[:npts][:,0]
    x2 = X[:npts][:,1]
    ax.scatter(x1, x2, s=0.5, c=col, zorder=1)
    plt.show()
```

```
visualize_polygons(p0,p1)
```

```
<ipython-input-104-b539349a2d90>:6: MatplotlibDeprecationWarning: Passing the closed parameter of __init__() positionally is
  polygon1 = Polygon(p0, True)
<ipython-input-104-b539349a2d90>:7: MatplotlibDeprecationWarning: Passing the closed parameter of __init__() positionally is
  polygon2 = Polygon(p1, True)
```



Please fill in all code blocks marked with a #TODO.

```
def threshold_activation1(x):
    """
    TODO: Implement one activation function (unit step function)

    Args:
      x (np.ndarray): input array

    Returns (np.ndarray): output array (with the same shape as input array)

    """
    # TODO:
    out = np.zeros(x.shape)
    for i, item in enumerate(x):
      if item >= 0:
        out[i] = 1
      else:
        out[i] = 0
    return out

def and_gate(x):
    """
    TODO: Implement an "AND" gate

    Args:
      x (np.ndarray): array with shape (n, 1), representing n neurons as inputs.

    Returns: (int): scalar of 1 or 0
    """
    # TODO:
    sum = 0
    for neuron in x:
      sum += neuron
    return (sum >= x.shape[0])
```

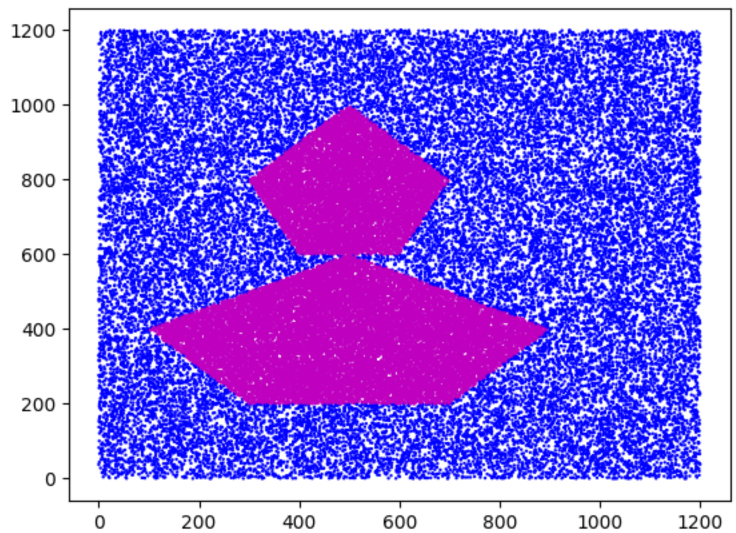
```
def or_gate(x):
    """
    TODO: Implement an "OR" gate

    Args:
      x (np.ndarray): array with shape (n, 1)

    Returns: (int): scalar of 1 or 0
    """
    # TODO:
    sum = 0
    for neuron in x:
      sum += neuron
    return 1 if sum >= 1 else 0

def analytical_parameters(p0, p1):
    """

    """
    ## Dimensionality
    x_dim = 2
    class_num = 2
    hidden_unit_num = 10
    # First Layer Parameter
    W = np.zeros((hidden_unit_num, x_dim))
    b = np.zeros((hidden_unit_num, 1))
    for i in range(5):
        # First polygon
        x1 = p0[i, 0]
        y1 = p0[i, 1]
        x2 = p0[(i+1)%5, 0]
        y2 = p0[(i+1)%5, 1]
        W[i, :] = [y1 - y2, x2 - x1]
        b[i, :] = x1 * y2 - x2 * y1
        # Second polygon
        x1 = p1[i, 0]
        y1 = p1[i, 1]
        x2 = p1[(i+1)%5, 0]
        y2 = p1[(i+1)%5, 1]
        W[i + 5, :] = [y1 - y2, x2 - x1]
        b[i + 5, :] = x1 * y2 - x2 * y1
    return W,b

def predict_output_v1(X, W, b):

    predictions = []
    for idx in range(data.shape[0]):
        x = np.reshape(X[idx, :], (2, 1))     # x.shape (2,1)
        # First layer
        # W.shape (10, 2), b.shape (10, 1)
        first_layer_output = np.matmul(W, x) + b # first_layer_output.shape (10, 1)
        first_layer_output = threshold_activation1(first_layer_output) #first_layer_output.shape (10, 1)
        # Second layer
        first_polygon = first_layer_output[0:5, :]
        second_polygon = first_layer_output[5:10, :]
        first_gate_output = and_gate(first_polygon)
        second_gate_output = and_gate(second_polygon)
        # Output layer
        input_to_final_gate = [first_gate_output, second_gate_output]
        prediction = or_gate(input_to_final_gate)
        predictions.append(prediction)
    return predictions

def predict_output_v2(X, W, b):
    """
    #TODO: Update usage of the gates in this function
    """
    ## Cache of Predictions
    predictions = []
    ## Cycle Through Data Points
    for idx in range(data.shape[0]):
        x = np.reshape(X[idx, :], (2, 1))
        # First layer
        first_layer_output = np.matmul(W, x) + b
        first_layer_output = threshold_activation1(first_layer_output)
        # Second layer
        first_polygon = first_layer_output[0:5, :]
```

```
        first_gate_output = and_gate(first_polygon)
        # Output layer
        prediction = or_gate(first_gate_output)
        predictions.append(prediction)
    return predictions

def calc_accuracy(true_y, pred_y):
    """

    """
    true_prediction_num = 0
    for i, py in enumerate(pred_y):
        if py == true_y[i]:
            true_prediction_num += 1
    accuracy = true_prediction_num / len(pred_y)
    print("Accuracy: ", accuracy)
    return accuracy
```

Sanity check: If you correctly implemented the 'and gate' and 'or gate', all points should be classified correctly when you make predictions using `predict_output_v1()`. You should provide the datapoint visualization plot and the accuracy in your report submission.

```
## Load Our Parameters
W, b = analytical_parameters(p0, p1)

## Make Predictions
pred_Y = predict_output_v1(X, W, b)

## Compute Accuracy
acc = calc_accuracy(Y, pred_Y)
assert (acc == 1)

## Visualize Predictions
visualize_datapoints(X, np.array(pred_Y))
```

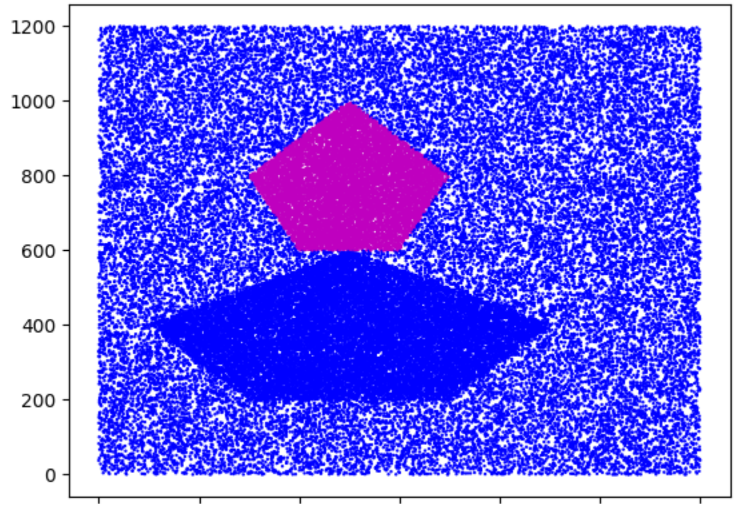
```
Accuracy:  1.0
```



In the code above, change the gates in `predict_output_v2()` such that only the points in the top polygon are classified correctly. Visualize your result, report the accuracy of this model, and attach it to the report submission.

To further clarify, you should **only** change the usage of the gating functions, not the code inside the gating function itself.

```
## Load Our Parameters
W, b = analytical_parameters(p0, p1)

## Make Predictions
pred_Y = predict_output_v2(X, W, b)

## Visualize Predictions
visualize_datapoints(X, np.array(pred_Y))
```



Problem 1b-d

Complete problems 1b through 1d in the space below. Please use markdown to clearly distinguish your answers for each part. Include appropriate visualizations generated here in your final report.

```
import torch
import torch.nn as nn
import random


## HyperParameters (Do not adjust for Question 1b)##

n_epoch = 500
n_seed = 5
lr = 1
batch_size = 5000

test_split = 1/6

layer_1_node = 10
layer_2_node = 2

#######################
layer_dims = [layer_1_node, layer_2_node]
seed_values = [random.randint(0, 10000) for _ in range(5)]
```

Complete the implementation of the MLP class and preprocess_data function below. Refer to the pytorch API to understand how a proper model (module) should be set up and initialized.

```
class MLP(nn.Module):
    """
    MLP class to create a multi-layer perceptron dynamically.

    Args:
        input_dim (int): The dimensionality of the input features.
        layers_dims (list of int): A list specifying the number of units in each hidden layer.
        output_dim (int): The dimensionality of the output.
        seed_value (int, optional): Random seed for reproducibility. If this is set to None, no manual seed is set.

    Attributes:
        layers (nn.ModuleList): A ModuleList to hold all the layers including input, hidden and output layers.
    """

    def __init__(self, input_dim, layers_dims, output_dim, seed_value=None):
        """
        Initialize MLP.
        """
        super(MLP, self).__init__()

        ## TODO:
        if seed_value is not None:
            torch.manual_seed(seed_value)
```

```
        self.layers = nn.ModuleList()
        self.batch_norms = nn.ModuleList()

        for i in range(len(layers_dims)):
          if i == 0: # input layer
            # 1/2
            self.layers.append(nn.Linear(input_dim, layers_dims[i]))
            self.batch_norms.append(nn.BatchNorm1d(layers_dims[i]))
            # 2/2
            self.layers.append(nn.Linear(input_dim, layers_dims[i]))
          else: # hidden layers
            self.layers.append(nn.Linear(layers_dims[i-1], layers_dims[i]))
          self.batch_norms.append(nn.BatchNorm1d(layers_dims[i]))

        self.layers.append(nn.Linear(layers_dims[-1], output_dim)) # output layer
        self._initialize_weights()


    def _initialize_weights(self):
        """
        Initialize the weights and biases of the model.
        """
        ## TODO:
        for layer in self.layers:
            nn.init.xavier_uniform_(layer.weight)
            nn.init.uniform_(layer.bias)

    def forward(self, x):
        """
        Forward pass through the network.

        Args:
            x (torch.Tensor): input tensor.

        Returns:
            torch.Tensor: output tensor.
        """
        # TODO:

        # Skip duplicate input layer
        for i, layer in enumerate(self.layers[1:-1], start=1):
            x = layer(x)
            x = self.batch_norms[i](x)
            x = torch.sigmoid(x)

        # Output layer
        x = self.layers[-1](x)
        x = torch.sigmoid(x)

        return x


from matplotlib.axis import YAxis
def preprocess_data(X, Y, test_split=1/6):
  """
  Base on your observation of the dataset, perform any necessary preprocessing steps given data X and label Y

  Args:
    X, Y (np.ndarry): input arrays
    test_split (float): proportion of data to use for test set (default is set to 1/6)

  Return:
    X_train, X_test, y_train, y_test (torch.Tensor): output tensor objects for training/testing.

  """

  # Note - If you plan to use additional functions, please define them as inner functions
  # under preprocess_data. This will allow us to export preprocess_data function and test
  # it thorough autograder properly. For instance:

  # ... def preprocess_data(X, Y, test_split):
  # ...
  # ...     def inner_func():
  # ...         print("Hello, World!")
  # ...
  # ...     inner_func()
```

```
    # Tips: For debugging purposes, it is a good practice to perform unit tests on your inner functions
    # before you place them under the preprocess_data function.

    # TODO:

    def normalize(arr):
      arr_min = np.min(arr, axis=0)
      arr_max = np.max(arr, axis=0)
      return (arr - arr_min) / (arr_max - arr_min)

    # Observation: X > 0 -> needs to be zero-centered
    X = X - np.mean(X, axis=0)

    # Normalize data
    X = normalize(X)
    Y = normalize(Y)

    # Shuffle data
    n = X.shape[0]
    train_max = int((1-test_split) * n)
    shuffled_indices = np.random.permutation(n)
    shuffled_test = shuffled_indices[train_max:]
    shuffled_train = shuffled_indices[:train_max]
    X_train = X[shuffled_train]
    X_test = X[shuffled_test]
    y_train = Y[shuffled_train]
    y_test = Y[shuffled_test]

    # Update from NumPy arrays to Tensors
    X_train = torch.FloatTensor(X_train)
    y_train = torch.FloatTensor(y_train)
    X_test = torch.FloatTensor(X_test)
    y_test = torch.FloatTensor(y_test)

    return (X_train, X_test, y_train, y_test)


  # Reload the data
  data = np.load(DATA_PATH)

  Y = data[:,2]
  X = data[:,0:2]

  X_train, X_test, y_train, y_test = preprocess_data(X, Y, test_split)
```

Implement the train loop (for a single run)

```
from torch.utils.data import DataLoader, TensorDataset, IterableDataset

def train(model,
          loss_f,
          optimizer,
          X_train,
          y_train,
          X_test=None,
          y_test=None,
          n_epoch=500,
          batch_size=None,
          seed_value=0):
    """
      The main function for model training.

      Args:
        model (torch.nn.Module): model to train
        loss_f (torch.nn.Module): loss function
        optimizer (torch.optim.Optimizer): optimizer
        X_train, y_train (torch.Tensor): training data
        X_test, y_test (torch.Tensor): test data
        n_epoch (int): number of epochs
        batch_size (int): size of the batch
        seed_value (int): random seed value

      Returns:
        .... (to be added by student)
```

```
    """

    # TODO: Complete the train function. You need to implement mini-batch training for this question.
    #
    # Tips: Perform proper sanity checks to ensure your inputs are reasonable. Keep track of important variables
    # (loss, accuracy) throughout the training loop. Print intermediate values reguarly to help you track if
    # your training is working as intended (so that if something is wrong you can terminate the process early
    # instead of going through all 5 runs.)

    def correct_pred(y_pred, y):
      y_pred_labels = [1.0 if val >= 0.5 else 0.0 for val in y_pred]
      correctness = [1.0 if y_pred_labels[i]==y[i] else 0.0 for i in range(len(y))]
      return sum(correctness)

    torch.manual_seed(seed_value) # initialize seed

    # Use all data if no batch size is given
    if batch_size is None:
        batch_size = len(X_train)

    # Load data into PyTorch
    train_data = TensorDataset(X_train, y_train)
    train_loader = DataLoader(train_data, batch_size=batch_size)

    # Create empty lists for loss and accuracy memory
    train_loss = []
    test_loss = []
    train_accuracy = []
    test_accuracy = []

    for epoch in range(n_epoch):
        model.train()  # Train the model for each epoch

        total_train_loss = 0.0
        correct_train = 0
        total_train = 0

        for X,y in train_loader:
            # Zero the gradients
            optimizer.zero_grad()
            model.zero_grad()

            # Forward pass
            y_pred = model(X).squeeze(1)
            y_pred = y_pred.float()
            # print('before:', y_pred.shape)
            # print('after:', y_pred.shape)
            loss = loss_f(y_pred, y)

            # Backpropagation and optimization
            loss.backward()
            optimizer.step()

            # Calculate total loss and accuracy for this batch
            total_train_loss += loss.item()
            total_train += y.size(0)
            correct_train += correct_pred(y_pred, y)

        # Calculate average training loss and accuracy for this epoch
        average_train_loss = total_train_loss / len(train_loader)
        train_acc = correct_train / total_train

        train_loss.append(average_train_loss)
        train_accuracy.append(train_acc)

        # If user defines test data
        if X_test is not None and y_test is not None:
            model.eval()  # Set the model in evaluation mode
            with torch.no_grad():
                # Forward pass
                ytest_pred = model(X_test).squeeze(1).float()
                test_loss_diff = loss_f(ytest_pred, y_test)

                # Calculate test accuracy
                correct_test = correct_pred(ytest_pred, y_test)
                test_acc = correct_test / len(y_test)
```

```
                test_loss.append(test_loss_diff.item())
                test_accuracy.append(test_acc)

        # Sanity check
        if (epoch + 1) % 20 == 0 or epoch == 0:
            print('Epoch:',epoch,'Train loss:', average_train_loss,'Train acc:',train_acc)

    return (train_loss, test_loss, train_accuracy, test_accuracy, ytest_pred)
```

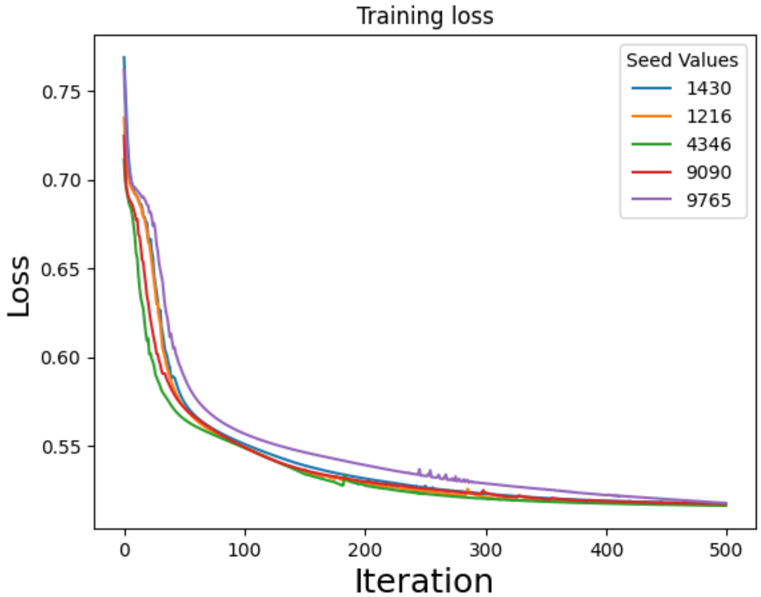
Now we start the training. We will iterate through 5 runs. To ensure reproducibility of the performance, we will be using the seed values to initialize our MLP and in our training loop. After the training, you should "**check if you can get the same model accuracy if a seed is re-used**".

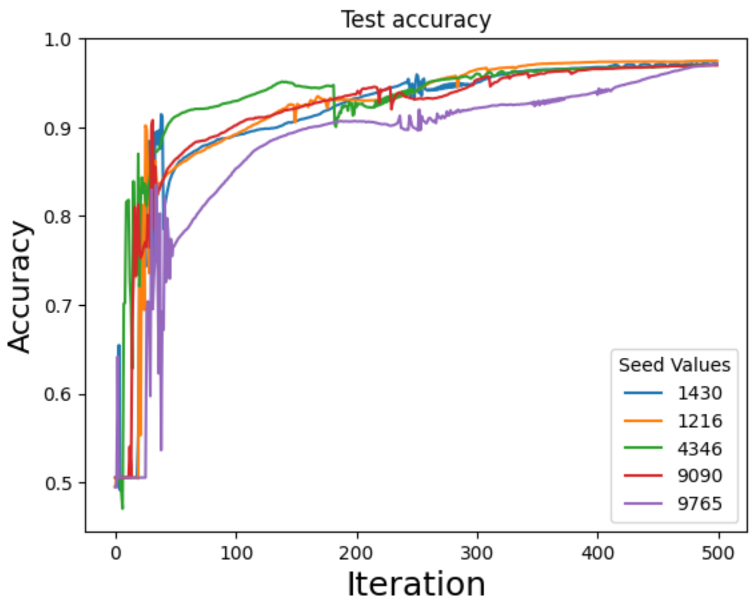
```
# TODO: complete the cell

# Create the model
input_dim = 2
output_dim = 1

# Training parameters
all_train_loss = []
all_train_acc = []
all_test_acc = []
test_predictions = []

## Iterate over Random Initializations
for idx in range(len(seed_values)):

  seed_value = seed_values[idx]
  print("~~ Beginning run {} with seed value {} ~~".format(idx, seed_value))

  # Train the model
  model = MLP(input_dim, layer_dims, output_dim, seed_value=seed_value)
  loss_f = nn.BCEWithLogitsLoss()  # Binary-cross-entropy loss
  optimizer = torch.optim.SGD(model.parameters(), lr=lr)  # Optimize with gradient descent
  train_loss, test_loss, train_accuracy, test_accuracy, prediction = train(model, loss_f, optimizer, X_train, y_train, X_test=X_t

  # Save data for next cell
  all_train_loss.append(train_loss)
  all_train_acc.append(train_accuracy)
  all_test_acc.append(test_accuracy)
  test_predictions.append(prediction)

# Calculate mean and standard deviation
mean_train_accuracy = np.mean(all_train_acc)
std_train_accuracy = np.std(all_train_acc)
mean_test_accuracy = np.mean(all_test_acc)
std_test_accuracy = np.std(all_test_acc)

# Output
print(f"Mean Train Accuracy: {mean_train_accuracy:.4f} (±{std_train_accuracy:.4f})")
print(f"Mean Test Accuracy: {mean_test_accuracy:.4f} (±{std_test_accuracy:.4f})")
```

```
~~ Beginning run 0 with seed value 1430 ~~
Epoch: 0 Train loss: 0.7688646972179413 Train acc: 0.50104
Epoch: 19 Train loss: 0.6757208168506622 Train acc: 0.50334
Epoch: 39 Train loss: 0.5878281652927398 Train acc: 0.88894
Epoch: 59 Train loss: 0.566882336139679 Train acc: 0.90094
Epoch: 79 Train loss: 0.5575132250785828 Train acc: 0.91046
Epoch: 99 Train loss: 0.5511491298675537 Train acc: 0.91848
Epoch: 119 Train loss: 0.5458098828792572 Train acc: 0.9277
Epoch: 139 Train loss: 0.5410597145557403 Train acc: 0.93526
Epoch: 159 Train loss: 0.5372219502925872 Train acc: 0.9421
Epoch: 179 Train loss: 0.5341253340244293 Train acc: 0.94698
Epoch: 199 Train loss: 0.5315989792346955 Train acc: 0.95068
Epoch: 219 Train loss: 0.5293839156627655 Train acc: 0.95398
Epoch: 239 Train loss: 0.5274693191051483 Train acc: 0.9573
Epoch: 259 Train loss: 0.5256397545337677 Train acc: 0.96044
Epoch: 279 Train loss: 0.5243007421493531 Train acc: 0.96254
Epoch: 299 Train loss: 0.5228508234024047 Train acc: 0.9659
Epoch: 319 Train loss: 0.5218447327613831 Train acc: 0.96746
Epoch: 339 Train loss: 0.5208738565444946 Train acc: 0.96916
Epoch: 359 Train loss: 0.5200937986373901 Train acc: 0.97096
Epoch: 379 Train loss: 0.5194573640823364 Train acc: 0.9722
Epoch: 399 Train loss: 0.5189340531826019 Train acc: 0.9733
Epoch: 419 Train loss: 0.5187453746795654 Train acc: 0.9732
Epoch: 439 Train loss: 0.518142569065094 Train acc: 0.97432
Epoch: 459 Train loss: 0.5179544806480407 Train acc: 0.97434
```

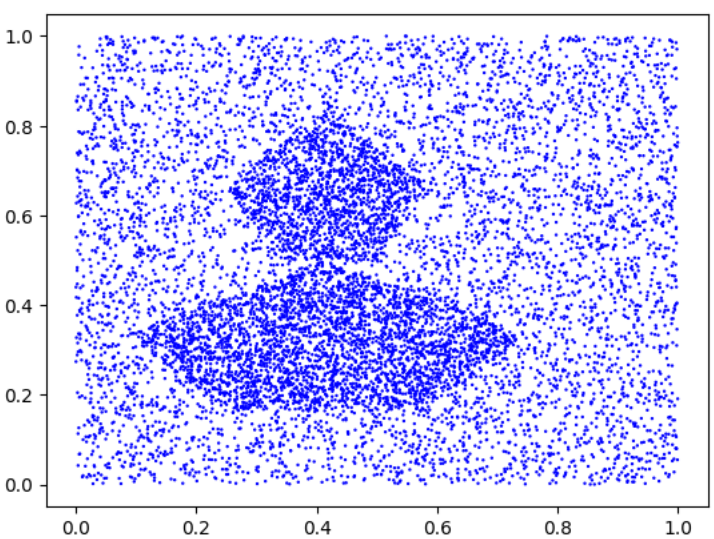
```
Epoch: 479 Train loss: 0.5177204072475433 Train acc: 0.97448
Epoch: 499 Train loss: 0.5174767374992371 Train acc: 0.97478
~~ Beginning run 1 with seed value 1216 ~~
Epoch: 0 Train loss: 0.7347791075706482 Train acc: 0.51164
Epoch: 19 Train loss: 0.6707484066486359 Train acc: 0.49896
Epoch: 39 Train loss: 0.5865094065666199 Train acc: 0.87412
Epoch: 59 Train loss: 0.5648634910583497 Train acc: 0.89816
Epoch: 79 Train loss: 0.5559177041053772 Train acc: 0.9093
Epoch: 99 Train loss: 0.5496491253376007 Train acc: 0.9186
Epoch: 119 Train loss: 0.5433067381381989 Train acc: 0.93148
Epoch: 139 Train loss: 0.5377150058746338 Train acc: 0.94176
Epoch: 159 Train loss: 0.5336088180541992 Train acc: 0.94794
Epoch: 179 Train loss: 0.5311920821666718 Train acc: 0.95156
Epoch: 199 Train loss: 0.528856760263443 Train acc: 0.9544
Epoch: 219 Train loss: 0.5270085155963897 Train acc: 0.95708
Epoch: 239 Train loss: 0.5252463042736053 Train acc: 0.96054
Epoch: 259 Train loss: 0.5236387014389038 Train acc: 0.96398
Epoch: 279 Train loss: 0.5221444129943847 Train acc: 0.96738
Epoch: 299 Train loss: 0.5211329638957978 Train acc: 0.96938
Epoch: 319 Train loss: 0.519766879081726 Train acc: 0.97198
Epoch: 339 Train loss: 0.5193441331386566 Train acc: 0.97224
Epoch: 359 Train loss: 0.5186060965061188 Train acc: 0.97382
Epoch: 379 Train loss: 0.5181577682495118 Train acc: 0.97444
Epoch: 399 Train loss: 0.5178127586841583 Train acc: 0.97478
Epoch: 419 Train loss: 0.5175144135951996 Train acc: 0.97512
Epoch: 439 Train loss: 0.5172404110431671 Train acc: 0.9755
Epoch: 459 Train loss: 0.5169805765151978 Train acc: 0.97574
Epoch: 479 Train loss: 0.516728812456131 Train acc: 0.97588
Epoch: 499 Train loss: 0.5164837598800659 Train acc: 0.97618
~~ Beginning run 2 with seed value 4346 ~~
Epoch: 0 Train loss: 0.7112266182899475 Train acc: 0.61948
Epoch: 19 Train loss: 0.6090987801551819 Train acc: 0.84412
Epoch: 39 Train loss: 0.5728463113307953 Train acc: 0.88578
```

```python
# TODO: Plot Results (Please plot the loss of all 5 runs in a same figure, and
# the accuracy of the runs in another figure). Use visualize_datapoints to check
# the performance of your model.

## From hw2
# Training loss
fig0=plt.figure(0)
plt.plot(all_train_loss[0],'-')
plt.plot(all_train_loss[1],'-')
plt.plot(all_train_loss[2],'-')
plt.plot(all_train_loss[3],'-')
plt.plot(all_train_loss[4],'-')
plt.legend(seed_values, title="Seed Values")
plt.xlabel('Iteration', fontsize=18)
plt.ylabel('Loss', fontsize=16)
plt.title('Training loss')
plt.show()
# Test accuracy
fig1=plt.figure(1)
plt.plot(all_test_acc[0],'-')
plt.plot(all_test_acc[1],'-')
plt.plot(all_test_acc[2],'-')
plt.plot(all_test_acc[3],'-')
plt.plot(all_test_acc[4],'-')
plt.legend(seed_values, title="Seed Values")
plt.xlabel('Iteration', fontsize=18)
plt.ylabel('Accuracy', fontsize=16)
plt.title('Test accuracy')
plt.show()

# Visualization
### not sure if i did this right
for i in range(len(test_predictions)):
  seed = seed_values[i]
  prediction = test_predictions[i]
  print("Run", i, "with seed",seed)
  visualize_datapoints(X_test, prediction)
```

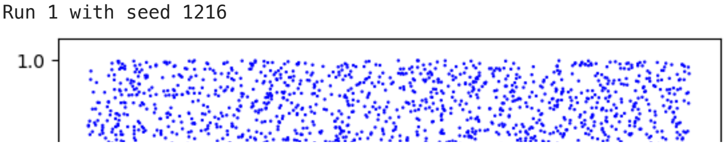


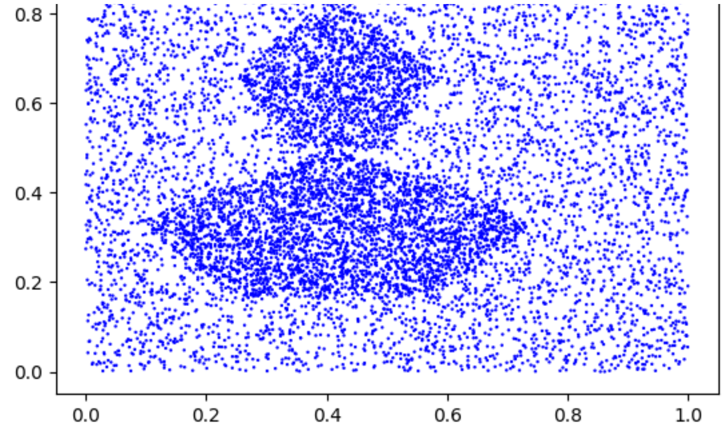



```
Run 0 with seed 1430
```

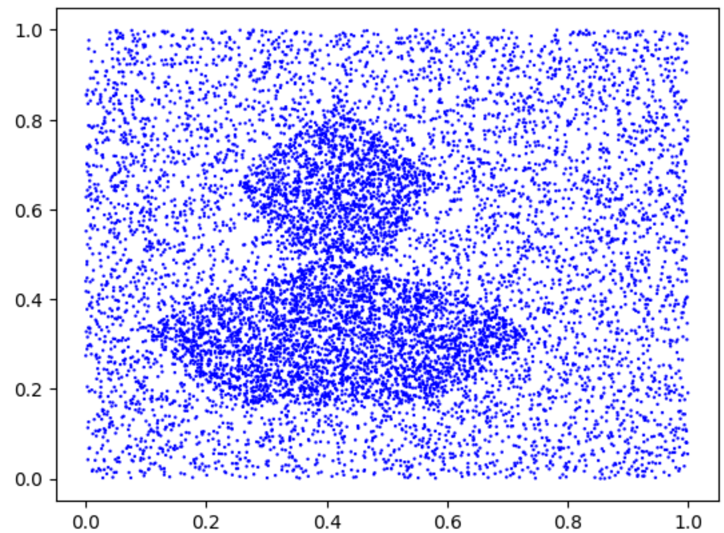


```
Run 1 with seed 1216
```

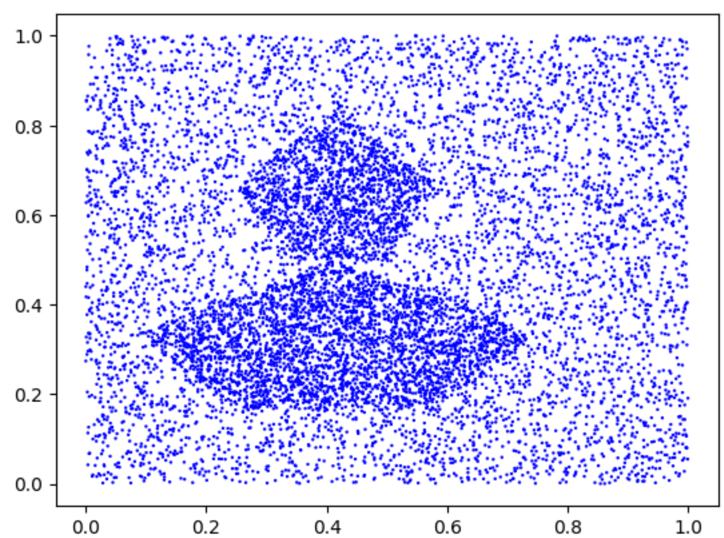




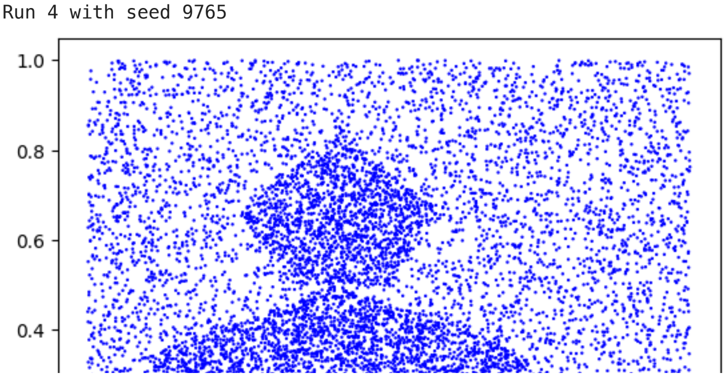

```
Run 2 with seed 4346
```




```
Run 3 with seed 9090
```




```
Run 4 with seed 9765
```



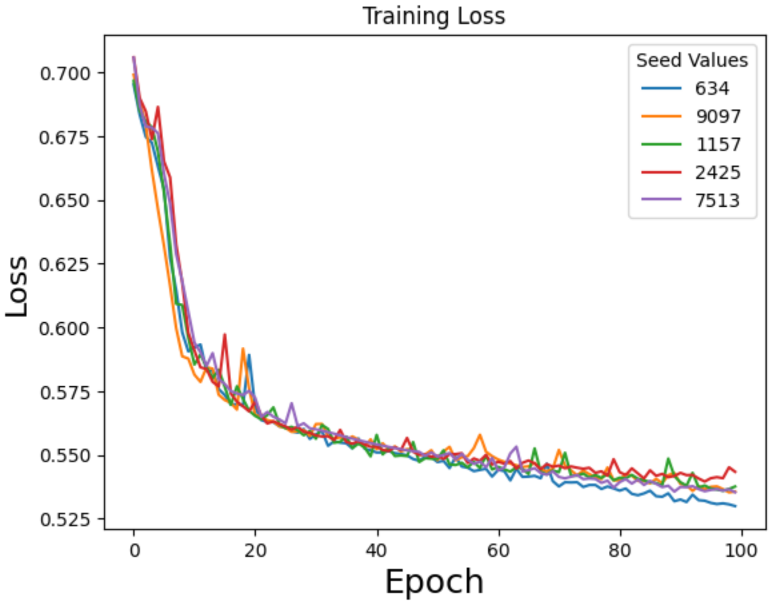
```
# Problem 1c: make adjustments to the layers, and then re-run the training loop with 5 runs and visualizations

## Hyperparameters

n_epoch = 500
n_seed = 5
lr = 1
batch_size = 5000

test_split = 1/6

layer_dims = []

##########################
seed_values = [random.randint(0, 10000) for _ in range(5)]


# Define a larger MLP model
class LargerMLP(nn.Module):
    def __init__(self, input_dim, hidden_layer_dims, output_dim, seed_value=None):

        super(LargerMLP, self).__init__()
        if seed_value is not None:
          torch.manual_seed(seed_value)

        self.layers = nn.ModuleList()
        self.batch_norms = nn.ModuleList()

        for i in range(len(hidden_layer_dims)):
            if i == 0:
                # Double input layers
                self.layers.append(nn.Linear(input_dim, hidden_layer_dims[i]))
                self.batch_norms.append(nn.BatchNorm1d(hidden_layer_dims[i]))
                self.layers.append(nn.Linear(input_dim, hidden_layer_dims[i]))
            else:
                self.layers.append(nn.Linear(hidden_layer_dims[i-1], hidden_layer_dims[i]))
            self.batch_norms.append(nn.BatchNorm1d(hidden_layer_dims[i]))
            self.layers.append(nn.Sigmoid())

        self.layers.append(nn.Linear(hidden_layer_dims[-1], output_dim))
        self.batch_norms.append(nn.BatchNorm1d(output_dim))
        self.layers.append(nn.Sigmoid())
        self._initialize_weights()

    def _initialize_weights(self):
        for layer in self.layers:
            if isinstance(layer, nn.Linear):
                nn.init.xavier_uniform_(layer.weight)
                nn.init.uniform_(layer.bias)

    def forward(self, x):
        count = 0
        for i, layer in enumerate(self.layers[1:], start=1):
            x = layer(x)
            if isinstance(layer, nn.Linear):
              count += 1
              x = self.batch_norms[count](x)

        return x

# Initialize parameters
input_dim = 2
hidden_layers = [50, 20, 10, 5]
output_dim = 1
train_accuracies = []
test_accuracies = []
test_predictions = []
train_losses = []
n_epoch = 100

# Train AND SAVE the model
for idx, seed in enumerate(seed_values):
    # Set output to start at run = 1 because that makes more sense
    print("~~ Beginning run {} with seed value {} ~~".format(idx+1, seed))
    torch.manual_seed(seed)

    # Train the larger model
```

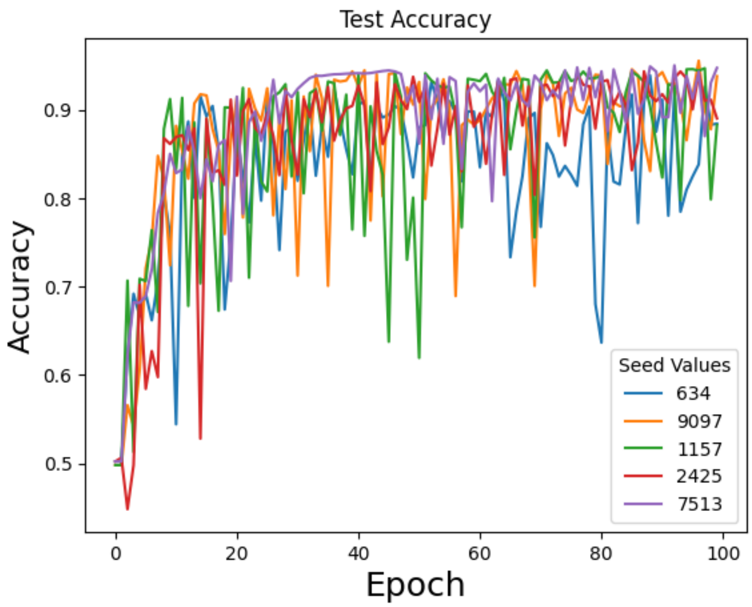
```
    larger_model = LargerMLP(input_dim, hidden_layers, output_dim)
    loss_f = nn.BCEWithLogitsLoss()
    optimizer = torch.optim.SGD(larger_model.parameters(), lr=lr)

    train_loss, test_loss, train_accuracy, test_accuracy, prediction = train(larger_model, loss_f, optimizer, X_train, y_train, X

    # Store accuracies
    train_losses.append(train_loss)
    train_accuracies.append(train_accuracy)
    test_accuracies.append(test_accuracy)
    test_predictions.append(prediction)

# Calculate mean and standard deviation
mean_train_accuracy = np.mean(train_accuracies)
std_train_accuracy = np.std(train_accuracies)
mean_test_accuracy = np.mean(test_accuracies)
std_test_accuracy = np.std(test_accuracies)

# Output
print("Larger MLP Depth and Width:")
print("Depth:", len(hidden_layers))
print("Width:", hidden_layers)
print(f"Mean Train Accuracy: {mean_train_accuracy:.4f} (±{std_train_accuracy:.4f})")
print(f"Mean Test Accuracy: {mean_test_accuracy:.4f} (±{std_test_accuracy:.4f})")
```

```
~~ Beginning run 1 with seed value 634 ~~
Epoch: 0 Train loss: 0.6952885568141938 Train acc: 0.62042
Epoch: 19 Train loss: 0.5892397999763489 Train acc: 0.83054
Epoch: 39 Train loss: 0.551787942647934 Train acc: 0.9221
Epoch: 59 Train loss: 0.5415468215942383 Train acc: 0.9425
Epoch: 79 Train loss: 0.5368973314762115 Train acc: 0.94394
Epoch: 99 Train loss: 0.5299693942070007 Train acc: 0.95944
~~ Beginning run 2 with seed value 9097 ~~
Epoch: 0 Train loss: 0.6991135597229003 Train acc: 0.59262
Epoch: 19 Train loss: 0.5760006248950958 Train acc: 0.8733
Epoch: 39 Train loss: 0.5561092019081115 Train acc: 0.91456
Epoch: 59 Train loss: 0.5495338261127471 Train acc: 0.92456
Epoch: 79 Train loss: 0.5400612592697144 Train acc: 0.94236
Epoch: 99 Train loss: 0.5356465756893158 Train acc: 0.94474
~~ Beginning run 3 with seed value 1157 ~~
Epoch: 0 Train loss: 0.6967183947563171 Train acc: 0.60852
Epoch: 19 Train loss: 0.5673565745353699 Train acc: 0.89544
Epoch: 39 Train loss: 0.5494532227516175 Train acc: 0.93328
Epoch: 59 Train loss: 0.544317102432251 Train acc: 0.93646
Epoch: 79 Train loss: 0.5401012897491455 Train acc: 0.93908
Epoch: 99 Train loss: 0.5376296579837799 Train acc: 0.93852
~~ Beginning run 4 with seed value 2425 ~~
Epoch: 0 Train loss: 0.7059306085109711 Train acc: 0.5868
Epoch: 19 Train loss: 0.5670938730239868 Train acc: 0.9002
Epoch: 39 Train loss: 0.5532082200050354 Train acc: 0.92184
Epoch: 59 Train loss: 0.5462502181529999 Train acc: 0.93258
Epoch: 79 Train loss: 0.5483429789543152 Train acc: 0.91698
Epoch: 99 Train loss: 0.5434133768081665 Train acc: 0.92864
~~ Beginning run 5 with seed value 7513 ~~
Epoch: 0 Train loss: 0.7057826459407807 Train acc: 0.57368
Epoch: 19 Train loss: 0.5751749396324157 Train acc: 0.87952
Epoch: 39 Train loss: 0.5546858549118042 Train acc: 0.91814
Epoch: 59 Train loss: 0.5488221049308777 Train acc: 0.92484
Epoch: 79 Train loss: 0.5372176766395569 Train acc: 0.94548
Epoch: 99 Train loss: 0.5353750288486481 Train acc: 0.94564
Larger MLP Depth and Width:
Depth: 4
Width: [50, 20, 10, 5]
Mean Train Accuracy: 0.8998 (±0.0731)
Mean Test Accuracy: 0.8615 (±0.0986)
```

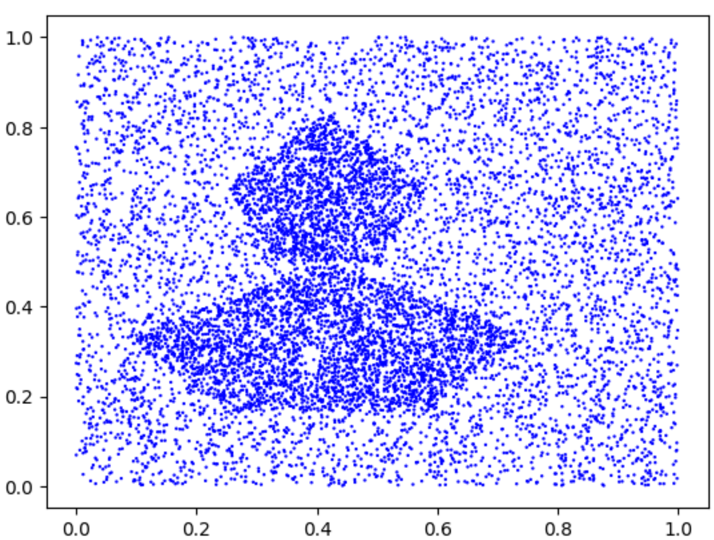
```
# Plots
# Training loss
fig0=plt.figure(0)
plt.plot(train_loss[0],'-')
plt.plot(train_loss[1],'-')
plt.plot(train_loss[2],'-')
plt.plot(train_loss[3],'-')
plt.plot(train_loss[4],'-')
plt.legend(seed_values, title="Seed Values")
plt.xlabel('Epoch', fontsize=18)
plt.ylabel('Loss', fontsize=16)
plt.title('Training Loss')
plt.show()
```

```
# Test accuracy
fig1=plt.figure(1)
plt.plot(test_accuracies[0],'-')
plt.plot(test_accuracies[1],'-')
plt.plot(test_accuracies[2],'-')
plt.plot(test_accuracies[3],'-')
plt.plot(test_accuracies[4],'-')
plt.legend(seed_values, title="Seed Values")
plt.xlabel('Epoch', fontsize=18)
plt.ylabel('Accuracy', fontsize=16)
plt.title('Test Accuracy')
plt.show()

# Visualization
for i in range(len(test_predictions)):
  seed = seed_values[i]
  prediction = test_predictions[i]
  print("Run", i, "with seed",seed)
  visualize_datapoints(X_test, prediction)
```

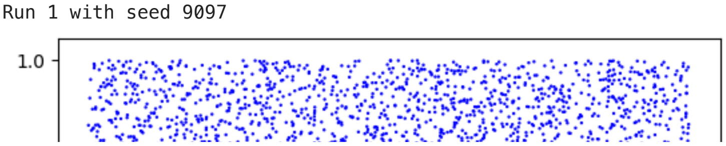


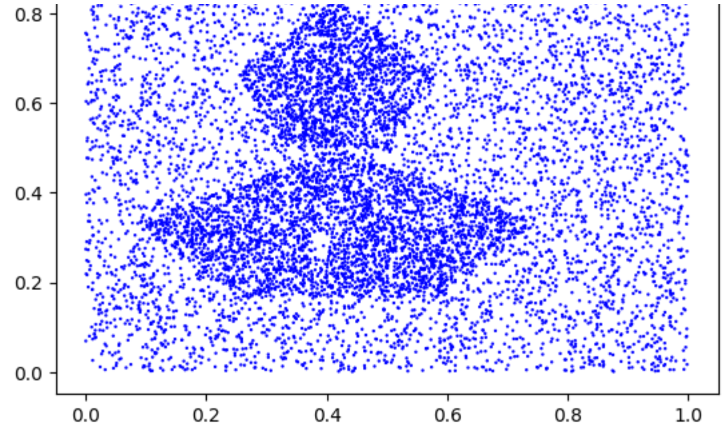



```
Run 0 with seed 634
```

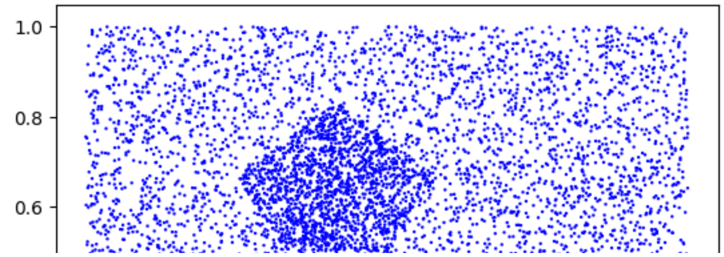


```
Run 1 with seed 9097
```





```
Run 2 with seed 1157
```



For Problem 1d, please write your response in the Latex report.

Problem 2

All code for Problem 2 should go below. We provide data loaders and relevant imports to get you started. If you are working locally (instead of using Google Colab), we recommend using Conda to install pytorch (https://pytorch.org).

Imports

```
Run 3 with seed 2425
```

```
## Additional External Libraries (Deep Learning)
import torch
import torch.nn as nn
from torch.autograd import Variable
from torch.utils.data import Dataset, DataLoader, SubsetRandomSampler
from torchvision import transforms as tfs
from PIL import Image
from torchvision.datasets import FashionMNIST
```

Data Loading

```
# Hyperparameter (Feel free to make modifications)
TRAIN_BATCH_SIZE = 50
VAL_BATCH_SIZE = 50
TEST_BATCH_SIZE = 1

# Transform data to PIL images
transforms = tfs.Compose([tfs.ToTensor()])

# Train/Val Subsets
train_mask = range(50000)
val_mask = range(50000, 60000)

# Download/Load Dataset
train_dataset = FashionMNIST('./data', train=True, transform=transforms, download=True)
test_dataset = FashionMNIST('./data', train=False, transform=transforms, download=True)

# Data Loaders
train_dataloader = DataLoader(train_dataset, batch_size=TRAIN_BATCH_SIZE, sampler=SubsetRandomSampler(train_mask))
val_dataloader = DataLoader(train_dataset, batch_size=VAL_BATCH_SIZE, sampler=SubsetRandomSampler(val_mask))
test_dataloader = DataLoader(test_dataset, batch_size=TEST_BATCH_SIZE)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to ./data/FashionMNIST/raw
```

```
100%|██████████| 26421880/26421880 [00:01<00:00, 16887144.64it/s]
Extracting ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/rav
100%|██████████| 29515/29515 [00:00<00:00, 296768.45it/s]
Extracting ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw/
100%|██████████| 4422102/4422102 [00:00<00:00, 4992187.81it/s]
Extracting ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/
100%|██████████| 5148/5148 [00:00<00:00, 13149985.99it/s]Extracting ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ./c
```

Problem 2a)

Design Model

```
class CNNet_2a(nn.Module):

    def __init__(self):
        """

        """
        ## Inherent Torch Module
        super(CNNet_2a, self).__init__()

        ##TODO: Initialize Model Layers
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2) # Halve the data
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(32 * 7 * 7, 128)  # Images are 28x28
        self.fc2 = nn.Linear(128, 10)  # Data has 10 classes
        self.relu = nn.ReLU()

    def forward(self, x):
        """

        """
        ##TODO: Setup Forward Pass
        x = self.conv1(x)
        x = self.relu(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.relu(x)
        x = self.pool2(x)
        x = x.view(x.size(0), -1)  # Flatten the output
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

Model Training

```
def train(model,
          loss_f,
          optimizer,
          n_epoch=50,
          train_dataloader=train_dataloader,
          val_dataloader=val_dataloader,
          test_dataloader=test_dataloader,
          seed_value=None):
    """

    """
    ##TODO: Implement training loop
```

```
    if seed_value is not None:
        # Set random seeds for reproducibility
        torch.manual_seed(seed_value)

    model.train()

    train_losses = []
    val_losses = []
    train_accuracies = []  # Store training accuracy
    val_accuracies = []  # Store validation accuracy

    for epoch in range(n_epoch):
        # Training
        train_loss = 0.0
        correct_train = 0
        total_train = 0
        model.train()
        for batch in train_dataloader:
            x, y = batch

            optimizer.zero_grad()
            y_pred = model(x)
            loss = loss_f(y_pred, y)
            loss.backward()
            optimizer.step()

            train_loss += loss.item()
            _, y_pred_binary = torch.max(y_pred.data, 1)
            total_train += y.size(0)
            correct_train += (y_pred_binary == y).sum().item()

        train_accuracy = correct_train / total_train
        train_accuracies.append(train_accuracy)
        train_losses.append(train_loss / len(train_dataloader))

        # Validation
        val_loss = 0.0
        correct_val = 0
        total_val = 0
        model.eval()
        with torch.no_grad():
            for batch in val_dataloader:
                x, y = batch

                y_pred = model(x)
                loss = loss_f(y_pred, y)
                val_loss += loss.item()
                _, y_pred_binary = torch.max(y_pred.data, 1)
                total_val += y.size(0)
                correct_val += (y_pred_binary == y).sum().item()

        val_accuracy = correct_val / total_val
        val_accuracies.append(val_accuracy)
        val_losses.append(val_loss / len(val_dataloader))

        print(f"Epoch {epoch + 1}/{n_epoch} - Train acc: {train_accuracy:.4f}, Val acc: {val_accuracy:.4f}, Train loss: {train_lo

    if test_dataloader is not None:
        test_loss = 0.0
        correct_test = 0
        total_test = 0
        model.eval()
        with torch.no_grad():
            for batch in test_dataloader:
                x, y = batch

                y_pred = model(x)
                loss = loss_f(y_pred, y)
                test_loss += loss.item()
                _, y_pred_binary = torch.max(y_pred.data, 1)
                total_test += y.size(0)
                correct_test += (y_pred_binary == y).sum().item()

        test_loss /= len(test_dataloader)
        test_accuracy = correct_test / total_test
        print(f"Test Loss: {test_loss:.4f}")
        print(f"Test Accuracy: {100 * test_accuracy:.2f}%")
```

```
    # Plot the training loss and validation accuracy over epochs
    plt.figure(figsize=(12, 4))
    plt.subplot(1, 2, 1)
    plt.plot(range(n_epoch), train_losses, label='Training Loss')
    plt.plot(range(n_epoch), val_losses, label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(range(n_epoch), train_accuracies, label='Training Accuracy')
    plt.plot(range(n_epoch), val_accuracies, label='Validation Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()

    plt.show()

    return train_losses, val_losses, train_accuracies, val_accuracies


## TODO: choose reasonable hyperparameters (feel free to make adjustments)
n_epoch = 50
model = CNNet_2a()
lr = 0.001
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
loss_f = nn.CrossEntropyLoss()


## Run Training Loop
train_losses, val_losses, train_accuracies, val_accuracies = train(model, loss_f, optimizer, n_epoch=n_epoch)
```

```
Epoch 1/50 - Train acc: 0.9976, Val acc: 0.9153, Train loss: 0.0069, Val loss: 1.0075
Epoch 2/50 - Train acc: 0.9973, Val acc: 0.9067, Train loss: 0.0091, Val loss: 1.0750
Epoch 3/50 - Train acc: 0.9973, Val acc: 0.9112, Train loss: 0.0090, Val loss: 1.0152
Epoch 4/50 - Train acc: 0.9986, Val acc: 0.9115, Train loss: 0.0039, Val loss: 1.0454
Epoch 5/50 - Train acc: 0.9970, Val acc: 0.9139, Train loss: 0.0091, Val loss: 1.0549
Epoch 6/50 - Train acc: 0.9979, Val acc: 0.9120, Train loss: 0.0066, Val loss: 1.0325
Epoch 7/50 - Train acc: 0.9976, Val acc: 0.9148, Train loss: 0.0079, Val loss: 0.9710
Epoch 8/50 - Train acc: 0.9983, Val acc: 0.9130, Train loss: 0.0060, Val loss: 1.0360
Epoch 9/50 - Train acc: 0.9979, Val acc: 0.9152, Train loss: 0.0070, Val loss: 1.1050
Epoch 10/50 - Train acc: 0.9970, Val acc: 0.9100, Train loss: 0.0107, Val loss: 1.1262
Epoch 11/50 - Train acc: 0.9985, Val acc: 0.9138, Train loss: 0.0049, Val loss: 1.0682
Epoch 12/50 - Train acc: 0.9992, Val acc: 0.9153, Train loss: 0.0033, Val loss: 1.0583
Epoch 13/50 - Train acc: 0.9985, Val acc: 0.8987, Train loss: 0.0061, Val loss: 1.2597
Epoch 14/50 - Train acc: 0.9956, Val acc: 0.9118, Train loss: 0.0159, Val loss: 1.0895
Epoch 15/50 - Train acc: 0.9988, Val acc: 0.9138, Train loss: 0.0043, Val loss: 1.0815
Epoch 16/50 - Train acc: 0.9973, Val acc: 0.9155, Train loss: 0.0101, Val loss: 1.0745
Epoch 17/50 - Train acc: 0.9980, Val acc: 0.9132, Train loss: 0.0067, Val loss: 1.2282
Epoch 18/50 - Train acc: 0.9976, Val acc: 0.9118, Train loss: 0.0087, Val loss: 1.1479
Epoch 19/50 - Train acc: 0.9980, Val acc: 0.9145, Train loss: 0.0072, Val loss: 1.1122
Epoch 20/50 - Train acc: 0.9972, Val acc: 0.9154, Train loss: 0.0102, Val loss: 1.0669
Epoch 21/50 - Train acc: 0.9985, Val acc: 0.9109, Train loss: 0.0043, Val loss: 1.2196
Epoch 22/50 - Train acc: 0.9977, Val acc: 0.9079, Train loss: 0.0079, Val loss: 1.2386
Epoch 23/50 - Train acc: 0.9973, Val acc: 0.9142, Train loss: 0.0078, Val loss: 1.1246
Epoch 24/50 - Train acc: 0.9979, Val acc: 0.9161, Train loss: 0.0078, Val loss: 1.1420
Epoch 25/50 - Train acc: 0.9983, Val acc: 0.9162, Train loss: 0.0047, Val loss: 1.1479
```

Problem 2b)

Now try to improve your model using additional techniques learned during class. You should be able to use the same training function as above, but will need to create a new model architecture.

Data Loading

You should maintain the splits from above, but feel free to alter the dataloaders (i.e. transforms) as you wish.

```
Epoch 37/50 - Train acc: 0.9974, Val acc: 0.9131, Train loss: 0.0087, Val loss: 1.2577
```

```
# Hyperparameter (Feel Free to Change These, but Make Sure your Training Loop Still Works as Expected)
TRAIN_BATCH_SIZE = 50
VAL_BATCH_SIZE = 50
TEST_BATCH_SIZE = 1

# Transform data to PIL images
transforms = tfs.Compose([
    tfs.ToPILImage(),
    tfs.Grayscale(num_output_channels=1),  # Ensure the images are grayscale
    tfs.Resize((28, 28)),  # Resize to 28x28
    tfs.ToTensor(),  # Convert to tensor
])

# Train/Val Subsets
train_mask = range(50000)
val_mask = range(50000, 60000)

# Download/Load Dataset
train_dataset = FashionMNIST('./data', train=True, transform=transforms, download=True)
test_dataset = FashionMNIST('./data', train=False, transform=transforms, download=True)

# Data Loaders
train_dataloader = DataLoader(train_dataset, batch_size=TRAIN_BATCH_SIZE, sampler=SubsetRandomSampler(train_mask))
val_dataloader = DataLoader(train_dataset, batch_size=VAL_BATCH_SIZE, sampler=SubsetRandomSampler(val_mask))
test_dataloader = DataLoader(test_dataset, batch_size=TEST_BATCH_SIZE)
```

Model Design

```
##TODO: Try to improve upon your previous architecture
# Add in dropout and batch normalization to prevent over-fitting
class CNNet_2b(nn.Module):

    def __init__(self):
        super(CNNet_2b, self).__init()

        # Convolution-pooling #1
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(16)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
```

```
        # C-P #2
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(32)  # Batch normalization after the second convolution
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Fully-connected layers
        self.fc1 = nn.Linear(32 * 7 * 7, 128)
        self.bn3 = nn.BatchNorm1d(128)  # Batch normalization after the first fully connected layer
        self.relu3 = nn.ReLU()
        self.dropout = nn.Dropout(0.5)  # Dropout with a 0.5 probability
        self.fc2 = nn.Linear(128, 10)  # 10 classes for FashionMNIST

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = self.relu2(x)
        x = self.pool2(x)
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = self.bn3(x)
        x = self.relu3(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

Model Training

```
##TODO: Fit and evaluate your model. What do you observe?
train_losses, val_losses, train_accuracies, val_accuracies = train(model, loss_f, optimizer, n_epoch=50)
```

```
Epoch 1/50 – Train acc: 0.9956, Val acc: 0.9124, Train loss: 0.0133, Val loss: 0.7220
Epoch 2/50 – Train acc: 0.9954, Val acc: 0.9146, Train loss: 0.0134, Val loss: 0.7416
Epoch 3/50 – Train acc: 0.9961, Val acc: 0.9147, Train loss: 0.0120, Val loss: 0.7133
Epoch 4/50 – Train acc: 0.9972, Val acc: 0.9126, Train loss: 0.0084, Val loss: 0.7304
Epoch 5/50 – Train acc: 0.9953, Val acc: 0.9124, Train loss: 0.0128, Val loss: 0.8278
Epoch 6/50 – Train acc: 0.9953, Val acc: 0.9164, Train loss: 0.0133, Val loss: 0.7011
Epoch 7/50 – Train acc: 0.9969, Val acc: 0.9092, Train loss: 0.0098, Val loss: 0.8181
Epoch 8/50 – Train acc: 0.9966, Val acc: 0.9106, Train loss: 0.0098, Val loss: 0.7603
Epoch 9/50 – Train acc: 0.9954, Val acc: 0.9144, Train loss: 0.0127, Val loss: 0.7426
Epoch 10/50 – Train acc: 0.9963, Val acc: 0.9109, Train loss: 0.0101, Val loss: 0.7895
Epoch 11/50 – Train acc: 0.9958, Val acc: 0.9156, Train loss: 0.0122, Val loss: 0.7455
Epoch 12/50 – Train acc: 0.9971, Val acc: 0.9151, Train loss: 0.0083, Val loss: 0.8073
Epoch 13/50 – Train acc: 0.9960, Val acc: 0.9170, Train loss: 0.0106, Val loss: 0.7663
Epoch 14/50 – Train acc: 0.9962, Val acc: 0.9125, Train loss: 0.0111, Val loss: 0.7969
Epoch 15/50 – Train acc: 0.9973, Val acc: 0.9108, Train loss: 0.0076, Val loss: 0.8358
Epoch 16/50 – Train acc: 0.9966, Val acc: 0.9121, Train loss: 0.0099, Val loss: 0.7861
Epoch 17/50 – Train acc: 0.9969, Val acc: 0.9152, Train loss: 0.0096, Val loss: 0.8040
Epoch 18/50 – Train acc: 0.9962, Val acc: 0.9148, Train loss: 0.0104, Val loss: 0.8592
Epoch 19/50 – Train acc: 0.9972, Val acc: 0.9107, Train loss: 0.0085, Val loss: 0.8142
Epoch 20/50 – Train acc: 0.9961, Val acc: 0.9101, Train loss: 0.0116, Val loss: 0.8480
Epoch 21/50 – Train acc: 0.9976, Val acc: 0.9122, Train loss: 0.0067, Val loss: 0.8728
Epoch 22/50 – Train acc: 0.9968, Val acc: 0.9126, Train loss: 0.0101, Val loss: 0.9047
Epoch 23/50 – Train acc: 0.9959, Val acc: 0.9109, Train loss: 0.0122, Val loss: 0.9226
Epoch 24/50 – Train acc: 0.9977, Val acc: 0.9092, Train loss: 0.0067, Val loss: 0.9868
Epoch 25/50 – Train acc: 0.9958, Val acc: 0.9155, Train loss: 0.0130, Val loss: 0.8725
Epoch 26/50 – Train acc: 0.9980, Val acc: 0.9164, Train loss: 0.0067, Val loss: 0.8477
Epoch 27/50 – Train acc: 0.9976, Val acc: 0.9101, Train loss: 0.0076, Val loss: 0.9148
Epoch 28/50 – Train acc: 0.9965, Val acc: 0.9123, Train loss: 0.0104, Val loss: 0.8606
Epoch 29/50 – Train acc: 0.9985, Val acc: 0.9117, Train loss: 0.0052, Val loss: 0.9490
Epoch 30/50 – Train acc: 0.9963, Val acc: 0.9156, Train loss: 0.0115, Val loss: 0.9000
```

Problem 2c)

Write your response in the Latex PDF report.

Generate hw4_utils.py file

Paste your code here to test it on autograder, this should include and_gate, or_gate, threshold_activation1, predict_output_v2, preprocess_data, MLP. This will create a file called hw4_utils.py. Note that even if some Errors show up in the autograder, it does not mean your code does not work. We will still look into your implementation manually.

```
%%writefile hw4_utils.py

# Paste your code here to test it on autograder, this should include and_gate, or_gate,
# threshold_activation1, predict_output_v2, preprocess_data, MLP. This will create a file
# called hw4_utils.py. Note that even if some Errors show up in the autograder, it does
# not mean your code does not work. We will still look into your implementation manually.

import numpy as np
import torch
import torch.nn as nn
import random

def threshold_activation1(x):

    out = np.zeros(x.shape)
    for i, item in enumerate(x):
      if item >= 0:
        out[i] = 1
      else:
        out[i] = 0
    return out

def and_gate(x):
    sum = 0
    for neuron in x:
      sum += neuron
    return (sum >= x.shape[0])


def or_gate(x):
    sum = 0
    for neuron in x:
      sum += neuron
    return 1 if sum >= 1 else 0
```

```
def predict_output_v2(X, W, b):
    ## Cache of Predictions
    predictions = []
    ## Cycle Through Data Points
    for idx in range(data.shape[0]):
        x = np.reshape(X[idx, :], (2, 1))
        # First layer
        first_layer_output = np.matmul(W, x) + b
        first_layer_output = threshold_activation1(first_layer_output)
        # Second layer
        first_polygon = first_layer_output[0:5, :]
        first_gate_output = and_gate(first_polygon)
        # Output layer
        prediction = or_gate(first_gate_output)
        predictions.append(prediction)
    return predictions

def preprocess_data(X, Y, test_split=1/6):
  def normalize(arr):
    arr_min = np.min(arr, axis=0)
    arr_max = np.max(arr, axis=0)
    return (arr - arr_min) / (arr_max - arr_min)

  # Observation: X > 0 -> needs to be zero-centered
  X = X - np.mean(X, axis=0)

  # Normalize data
  X = normalize(X)
  Y = normalize(Y)

  # Shuffle data
  n = X.shape[0]
  train_max = int((1-test_split) * n)
  shuffled_indices = np.random.permutation(n)
  shuffled_test = shuffled_indices[train_max:]
  shuffled_train = shuffled_indices[:train_max]
  X_train = X[shuffled_train]
  X_test = X[shuffled_test]
  y_train = Y[shuffled_train]
  y_test = Y[shuffled_test]

  # Update from NumPy arrays to Tensors
  X_train = torch.FloatTensor(X_train)
  y_train = torch.FloatTensor(y_train)
  X_test = torch.FloatTensor(X_test)
  y_test = torch.FloatTensor(y_test)

  return (X_train, X_test, y_train, y_test)

class MLP(nn.Module):

    def __init__(self, input_dim, layers_dims, output_dim, seed_value=None):
        super(MLP, self).__init__()
        if seed_value is not None:
            torch.manual_seed(seed_value)

        self.layers = nn.ModuleList()
        self.batch_norms = nn.ModuleList()

        for i in range(len(layers_dims)):
          if i == 0: # input layer
            # 1/2
            self.layers.append(nn.Linear(input_dim, layers_dims[i]))
            self.batch_norms.append(nn.BatchNorm1d(layers_dims[i]))
            # 2/2
            self.layers.append(nn.Linear(input_dim, layers_dims[i]))
          else: # hidden layers
            self.layers.append(nn.Linear(layers_dims[i-1], layers_dims[i]))
          self.batch_norms.append(nn.BatchNorm1d(layers_dims[i]))

        self.layers.append(nn.Linear(layers_dims[-1], output_dim)) # output layer
        self._initialize_weights()


    def _initialize_weights(self):
        for layer in self.layers:
            nn.init.xavier_uniform_(layer.weight)
            nn.init.uniform_(layer.bias)
```

```
            nn.init.uniform_(layer.bias)

    def forward(self, x):
        # Skip duplicate input layer
        for i, layer in enumerate(self.layers[1:-1], start=1):
            x = layer(x)
            x = self.batch_norms[i](x)
            x = torch.sigmoid(x)

        # Output layer
        x = self.layers[-1](x)
        x = torch.sigmoid(x)

        return x
```

```
Overwriting hw4_utils.py
```